

AD-A142 142

VLSI RESEARCH

12

SEMI-ANNUAL  
TECHNICAL R&D STATUS REPORT  
OCTOBER 1982 - APRIL 1984

PRINCIPAL INVESTIGATOR

R.W. BRODERSEN  
(415-542-1779)

FACULTY RESEARCHERS

R.W. BRODERSEN  
N. CHEUNG  
A. DESPAIN  
D.A. HODGES  
C. HU  
R. KATZ  
P. KO  
D. MESSERSCHMITT  
R.S. MULLER  
A. NEUREUTHER  
A.R. NEWTON  
W.G. OLDHAM  
J. OUSTERHOUT  
D.A. PATTERSON  
A. SANGIOVANNI-VINCENTELLI  
C. SEQUIN

DTIC  
SELECTED  
JUN 12 1984  
E

SPONSORED BY

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY  
ARPA ORDER NO. 4031

MONITORED BY NAVAL ELECTRONIC SYSTEMS COMMAND  
UNDER CONTRACT NO. N00039-C-0107

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

This document has been approved for public release and sale; its distribution is unlimited.

DTIC FILE COPY

**ELECTRONICS RESEARCH LABORATORY**  
**College of Engineering**  
**University of California, Berkeley, CA 94720**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
AD-A142 142		
4. TITLE (and Subtitle) VLSI Research		5. TYPE OF REPORT & PERIOD COVERED Technical R&D Status Report October 1983 - April 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R.W. Brodersen, et al		8. CONTRACT OR GRANT NUMBER(s) N00039-C-0107
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of California, Berkeley 404 Cory Hall Berkeley, CA 94720		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE
		13. NUMBER OF PAGES 1,500 approximately
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Department of Navy Naval Electronic Systems Command Washington, D.C. 20363		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved For Public Release Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		

DTIC  
ELECTE  
JUN 12 1984



VLSI RESEARCH  
SEMI-ANNUAL  
TECHNICAL R&D STATUS REPORT  
OCTOBER 1983 - APRIL 1984

PRINCIPAL INVESTIGATOR

R.W. BRODERSEN  
(415-542-1779)

FACULTY RESEARCHERS

R.W. BRODERSEN  
N. CHEUNG  
A. DESPAIN  
D.A. HODGES  
C. HU  
R. KATZ  
P. KO  
D. MESSERSCHMITT  
R.S. MULLER  
A. NEUREUTHER  
A.R. NEWTON  
W.G. OLDHAM  
J. OUSTERHOUT  
D.A. PATTERSON  
A. SANGIOVANNI-VINCENTELLI  
C. SEQUIN



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## TABLE OF CONTENTS

I. Executive Overview

II. Summary of Research

III. Publications

a. Architecture

b. Computer Aids for Design and Layout

c. Circuit & System Design

d. Technology

23

## Executive Overview

This report covers the period from October 1983 to April 1984 on contract No. N00039-C-0107. A few of the highlights of this report follow.

A scaled version of the RISC II chip has been fabricated and tested and these new chips have a cycle time that would outperform a VAX 11/780 by about a factor of two on compiled integer C programs. The architectural work on a RISC chip designed for a Smalltalk implementation has been completed. This chip, called SOAR (Smalltalk on a RISC), should run programs 4-15 times faster than the Xerox 1100 (Dolphin), a TTL minicomputer, and about as fast as the Xerox 1132 (Dorado), a \$100,000 ECL minicomputer.

The 1983 VLSI tools tape has been converted for use under the latest UNIX release (4.2). The Magic (formerly called Caddy) layout system will be a unified set of highly automated tools that cover all aspects of the layout process, including stretching, compaction, tiling and routing. A multiple window package and design rule checker for this system have just been completed and compaction and stretching are partially implemented. New slope-based timing models for the Crystal timing analyzer are now fully implemented and in regular use. In an accuracy test using a dozen critical paths from the RISC II processor and cache chips it was found that Crystal's estimates were within 5-10% of SPICE's estimates, while being a factor of 10,000 times faster.

A new approach to the state assignment problem which allows minimization of Finite State Machines has been developed. This work coupled with some new advances in two level logic minimization (ESPRESSO II), will provide the basis for a highly effective tool for FSM synthesis. The SPLICE 1.7 program which has been shown to have more than 2 orders of magnitude speed-up over SPICE while giving answers that are as accurate is now at over a 100 sites. The implementation of this approach onto a multiprocessor machine has been investigated (the BBN Butterfly machine) and a 70% efficiency for a 10 processor machine was found realizable (ie. 70% of the possible 10 times improvement over a uniprocessor was achieved).

A design study of VLSI communications has been completed which discusses fault tolerance, distribution of ports, routing and buffering schemes and self testing. Also novel mappings of digital filter architectures onto multiprocessor systems has been discovered which allow arbitrarily high sampling rates with a fixed speed technology.

A design frame for the Multibus system bus has been developed and a frame chip and printed circuit board have been integrated. This should allow "run time support" and "system level services" for a chip designer in testing and using his chip. The design frame chip and custom designed printed circuit board have been operating in a Sun workstation under the UNIX operating system.

A 1000 word speech recognition board, which uses two special purpose chips, has been successfully operated inside a SUN workstation. The UNIX drivers which allow direct interaction between the SUN cpu and the cpu on the recognition board have been written. The design of the board has been transferred to SRI, where one duplicate board has been made and about 10 more are planned in the near future.

The software system which performs the complete silicon compilation of digital filter banks from high level filter descriptions is now being transferred into industry. The complete generation of a 20,000 transistor circuit, including real time testing of the algorithm, can be performed within one day. A number

- 4 -

of these circuits have been fabricated and tested and they have all been found to meet specifications.

A single-chip full duplex linear predictive vocoder (LPC) circuit has been designed and tested. Some of the chips have been delivered to a company (GE) to perform critical evaluations of the speech quality when configured into an LPC-10 system.

A quantitative model for CMOS latch-up has been developed which has led to the development of a new technique for suppressing it. The SIMPL-1 program has been completed which uses a file of process steps and the CIF layout information to generate a " scanning electron microscope " like view of the device topography. A series of models have been developed aimed at the IC designer which explain the effects of hot electrons in a scaled technology.



## 1. ARCHITECTURE

### 1.1. Rounding up the RISC Project (D. Patterson, C. Séquin)

A shrunk version of the RISC II chip, implemented by straight-forward scaling down of the mask geometry to a Lambda of 1.5 microns, has been tested and evaluated. As expected, it was functionally correct since it came from the same CIF file; but the pleasant surprise was that it ran 50% faster than the previous version with Lambda equal to 2 microns. No simulation had been done for the new device geometry, and the fact that it performs so well is a tribute to the ruggedness of the RISC circuit design. It also proves, that within limits the straight-forward Mead-Conway scaling is indeed practical. The new chips, running at a 330ns cycle time (VDD=5V, VBB=VSS=0V, room temperature) would outperform a VAX 11/780 by about a factor of two on compiled integer C programs. These results have been presented at ISSCC last month [1].

This brings the RISC project to a close. Manolis Katevenis finished his PhD in November 1984, and Robert Sherburne will finish his PhD within a month. The two theses document the experiences gained in the RISC project. Katevenis' thesis starts from an analysis of inner loops of typical programs to determine the most frequently used operations and derives the implications for the selection of an instruction set and for the choice of features to be supported in hardware on a single-chip RISC. From this starting point it derives the RISC microarchitecture and discusses the associated trade-offs.

Sherburne's thesis focuses on the circuit design in RISC II. It emphasizes the fact that the circuit design has to be seen in context of the microarchitectural tradeoffs and can even influence the choice of the instruction set. In particular, it discusses the optimum size of the register file. Beyond a certain point, more registers are not necessarily better, since the longer busses will slow down the overall machine cycle. A marginal increase in the hit ratio in accessing scalar operands is not worth a slow-down of all instructions.

The optimal replacement strategy for swapping windows in the RISC register file has been analyzed in a paper published in IEEE Transactions on Computers [5].

### 1.2. Architecture for Software Prototyping (D. Patterson, D. Hodges)

We have completed the initial versions of the architecture simulator and Smalltalk-80 compiler for SOAR (Smalltalk On A RISC), our software prototyping architecture. We have recently run 8 small Smalltalk-80 benchmarks on the simulator and found promising results. A 800 ns cycle SOAR will run these small programs 4 to 15 times faster than the Xerox 1100 (Dolphin), a TTL minicomputer, and run between .3 to 1.5 times the speed of the Xerox 1132 (Dorado), an ECL minicomputer costing over \$100,000. Our next step in performance analysis will be to complete the other 40 "micro" benchmarks, and then run the dozen large Smalltalk benchmarks. These large benchmarks will give accurate predictions of the performance of SOAR, but we are very encouraged by these early estimates.

The next step will be to complete the layout, simulation, and timing verification to estimate accurately the SOAR cycle time.

## References:

- (1) R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, and C.H. Séquin: " A 32b NMOS Microprocessor with a Large Register File " , *ISSCC 1984, Digest of Tech. Papers*, San Francisco, Feb. 1984, pp 168-169.
- (2) M.G.H. Katevenis, " Reduced Instruction Set Computer Architecture for VLSI " , Ph.D thesis, U.C. Berkeley, Oct. 1983.
- (3) R.W. Sherburne, " Processor Design Tradeoffs in VLSI " , Ph.D thesis, U.C. Berkeley, (~ April, 1984).
- (4) R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, and C.H. Séquin, " Local Memory in RISCs " , *Proc. Intl. Conf. on Computer Design*, New York, Oct. 31 -Nov. 3, 1983, pp 149-152.
- (5) Y. Tamir and C.H. Séquin: " Strategies for Managing the Register File in RISC " , *IEEE Trans. on Computers*, C-23, No 11, Nov. 1983, pp 977-989.

### 1.3. Novel, High-Performance Architectures (S. Baden, A. Despain)

We are studying the problems of multiprocessor system design. In particular, the partitioning and synchronization problems that arise when many processors cooperate on dynamically changing computational structures. Such structures occur in very difficult calculations that today are only attempted on the largest machines.

We have begun evaluating benchmark programs that will be used to generate trace tapes on a Cray-1. We will use these tapes to measure the dynamic resource demands of the benchmarks (such as how many processes could be executed in parallel if there were sufficient processing elements available?) in order to refine the design of our multiprocessor architecture. Later, the tapes will also be used in trace-driven simulations of our machine design.

At the moment we are looking at a method for solving non-linear Partial Differential Equations (PDEs), in the presence of strong shocks (and other local irregularities), called *Adaptive Mesh Refinement* (AMR). AMR is attractive because it can be used to achieve more accurate results while incurring minimal space and time penalties. As it is representative of a much larger class of problems that cannot be statically partitioned, it is of particular interest to us.

Basically, AMR works by refining the solution grid wherever the solution is changing 'too' rapidly (as indicated by a Richardsonian error estimate) and by operating on a reduced time scale in those regions. Thus, the solver is able to distribute the computer's resources (memory words and processor cycles) where they are needed the most, rather than distributing them uniformly over the entire solution grid. Owing to the the dynamic nature of this algorithm, i.e. refined grids can shrink, grow, and disappear unpredictably, it is not possible to partition an AMR solver at compile time. Thus, to avoid excessive serial execution bottlenecks and communications delays, a scientific multiprocessor must provide run time support for partitioning and load balancing activities.

Recently published results support our hypothesis: in their investigation of an adaptive, parallel, finite element system (FEARS), simulated on two different multiprocessor architectures (cm\* and ZMOB, 256 Z-80's connected on a ring), Zave and Cole noted that serial bottlenecks accounted for 80% of the total execution time, and communication only 1%. Maximum speedups of only 3-5 were

measured. These results aren't surprising: in FEARS no attempt is made to partition newly generated subgrids (as in AMR, multi-level grid structures are used for improved accuracy), so if some were much larger than the others, then a small number of processors would be doing most of the work, and not communicating very often. The results of the FEARS study were not conclusive and we will test our hypothesis out by extracting the distribution of subgrid sizes from the Cray trace tapes.

Adaptive partitioning has not received much attention except in classic dataflow. True, partitioning is trivial in this instance and this has been cited as a major reason for adopting a dataflow-style architecture. But owing to the great cost of synchronizing low level scalar operations (among other difficulties) classic dataflow is unsuitable for applications like AMR; instead, high level, macro operations have to be used (i.e., vector add). It is for this reason that we believe that the problem of higher-level adaptive partitioning cannot be avoided, even if a non-traditional architecture were to be used.

We envision using a data-driven model of execution for our machine design: the nodes perform functions at a much higher level than classic dataflow (indeed they will be traditional, Cray-1-like arithmetic and logical functional units), arcs are bidirectional and have storage, and more complex firing rules are used to admit the processing of data-streams. In such a system it would be possible to exploit both the advantages of dataflow (dynamic detection of concurrency) as well as the advantage of control flow (efficiency of low-level operations such as vector arithmetic).

We believe that adaptive numerical techniques will become more prevalent in the future, owing to the emergence of commercial multiprocessors (i.e. the BBN Butterfly Machine, the Denelcor HEP-1, and the Cray-XMP). Unless the problem of adaptive partitioning is well understood, multiprocessors, whether they be of a traditional design or not, will not be cost-effective for the novel numerical methods.

#### **1.4. Multiprocessor Circuit Simulation (D.G. Messerschmitt)**

The project in which scheduling algorithms were developed for concurrent execution of the Doolittle LU decomposition algorithm has been completed. These scheduling algorithms have specific application to LU decomposition in circuit simulation, and more generally to the concurrent execution of a program on multiple processors where the communication delay between processors is significant in comparison to the computation time. Two types of scheduling heuristics were developed and compared: local algorithms and global algorithms. In the local algorithms, Hu's level scheduling algorithm was modified to attempt to find the best mapping between ready tasks and available processors to minimize communication delay. In the global algorithms, the remaining longest path was mapped onto a single processor in order to minimize the effect of communication delay on this critical path. Significant speedup in using these algorithms, often greater than 50%, was obtained on the LU decomposition of sparse matrices extracted from SPICE simulations of actual circuits. This work has been reported in a thesis [1] and papers are in preparation.

As reported six months ago, a method of achieving arbitrarily high sampling rates in IIR digital filtering with a fixed speed of hardware has been discovered. This technique is reported in a recent thesis by Lu [2], and will be reported in a

paper in the near future. This technique has been extended to an arbitrary filter (previously only simple poles were allowed). In addition, an approach proposed by Rao and Kailath at Stanford in which good numerical properties are maintained by using an orthogonal state matrix has been extended to allow high sampling rates. Some preliminary work has been done on the partitioning of this implementation into individual chips.

The project in which automatic generation of topology for interconnection of multiple processors for a specific applications domain is nearing completion. Methods of measuring the performance of a given interconnection which are hopefully simple yet related to program speedup are being examined. A thesis in this area should be completed within six months.

Finally, work has started in the design of algorithms and multiprocessor architectures for implementation of the finite element method widely used in seismic and structural modelling.

#### References:

- (1) W-H Yu, " LU Decomposition on a Multiprocessing System with Communication Delay " , Ph.D. Thesis, University of California, Berkeley, CA., March 1984.
- (2) H-H Lu, " High Speed Recursive Filtering " , Ph.D. Thesis, University of California, Berkeley, Ca., Dec. 1983.

#### 1.5. VLSI Multicomputers (C. Sequin)

The design study of a VLSI communications component has come to a conclusion with the completion of the PhD thesis by Richard Fujimoto [1]. This thesis discusses the design of VLSI components with about 100,000 devices that would permit the construction of VLSI multicomputer networks which communicate through dedicated links between nearest neighbors.

It is shown that, when considering a fixed maximum output bandwidth from a single-chip component, it is preferable to have only a few ports with as much bandwidth each as possible. The higher bandwidth per port almost always more than compensates for the larger number of node-to-node hops in a network with correspondingly lower branching. In addition, the tradeoffs among different routing and buffering schemes are also analyzed. Many crucial results were obtained with the help of SIMON [2], a simulator for such closely coupled multi-computer systems running on a VAX under UNIX. Fujimoto, now a professor at Utah, will continue to work in that area.

In addition to high performance, high *reliability* is another strong reason to consider multicomputer systems. When executing large computation tasks, such as those involved in weather forecasting or wind-tunnel simulation, the system may have to operate continuously for many hours. Due to various degenerative processes and due to our inability to completely test VLSI chips, there is a high probability that a component of the computing system will generate incorrect results during the execution of large tasks. We are studying ways to prevent such component failure from leading to system failure through the use of *fault-tolerance techniques*. In particular, we are exploring the use of techniques that involve a small performance penalty and do not require significant increases in the complexity of the system.



A multicomputer is especially well suited for fault-tolerance techniques since it is partitioned into independent and "intelligent" components (the nodes). Fault-free components can adapt to changes in faulty components and continue their operation in a way that leads to correct system output despite the fault. Detecting errors immediately after they occur, greatly simplifies the error recovery procedures that must be invoked in order to restore the system to a valid state. This is accomplished through the use of *self-checking nodes* that signal an error to their neighbors when they produce incorrect results. With VLSI, an effective way to implement a self-checking node is by using duplicate functional modules whose outputs are continuously compared. Such a *duplication and matching* scheme at the processor level [3] also has the advantage of conceptual simplicity.

The critical circuit in this scheme is a comparator which must not be susceptible to faults that can remain undetected and later mask the failure of the functional modules. Since physical defects can affect the comparator, it must be *self-testing* so that it produces an error indication when it incurs such a defect. Based on a new fault model for PLA's we have shown that with both NMOS and CMOS technologies a PLA can be used to implement such a comparator. The design of such a comparator and its behavior under most conceivable faults that are likely to occur in a VLSI system are analyzed in [4]. The analysis even for this simple and regular component is rather difficult and tedious; because of that, the duplication and matching scheme looks particularly attractive since it requires such a detailed analysis for only one component: the self-checking comparator.

## References

- (1) R.M. Fujimoto, " VLSI Communication Components for Multicomputer Networks " , PhD Thesis, U.C. Berkeley, Fall 1983.
- (2) R.M. Fujimoto, " SIMON, A simulator of Multicomputer Networks " , Tech. Report No. UCB/CSD 83/140, Sept. 1983.
- (3) Y. Tamir and C.H. Sequin, " Self-checking VLSI Building Blocks for Fault-Tolerant Multicomputers " , *Proc. Intl. Conf. on Computer Design*, New York, Oct. 31 -Nov. 3, 1983, pp 561-564.
- (4) Y. Tamir and C.H. Sequin, " Design and Application of Self-Testing Comparators Implemented with MOS PLAs " , scheduled for publication in *IEEE Transaction on Computers*, Spring 1984.

## **2. COMPUTER AIDS FOR DESIGN AND LAYOUT**

### **2.1. 1983 VLSI Tools Distribution, 4.2 version (J. Ousterhout)**

The 1983 tools tape, which we have been distributing since April 1983, has been upgraded to run under the 4.2 version of Berkeley Unix (the initial version of the tape runs only under version 4.1). Except for the switchover to 4.2, there are no major changes to the programs. The new tape has been available since early February 1984.

### **2.2. The Magic Layout System (J. Ousterhout)**

Magic is a new VLSI layout system that has been under development for about a year (until recently, it was called "Caddy"). In the last six months we have completed the implementation of the multiple-window package and the design-rule checker. The window package allows a number of overlapping windows of different types to coexist on the color display. Different windows may contain different views on the same circuit or views on different circuits. Information can be copied from one window to another. Windows can also be used for other functions such as menus, a glyph editor, or a color map editor.

Magic's design-rule checker is an incremental one that runs in background to update error information as soon as possible after the circuit has changed. Magic records areas that have been modified and remembers this information until the areas have been re-checked, even if this doesn't happen until a later editing session. For small changes, the re-check occurs instantaneously. For large changes, such as moving a large cell so that it overlaps another large cell, more time may be required. Early measurements indicate that the checker can process about 800 tiles/second when working entirely within one cell, or about 200 tiles/second when registering information from overlapping subcells.

Compaction and stretching are provided with a new operation called "plowing", which allows portions of the circuit to be re-arranged while maintaining the design rules and connectivity. The plowing design was completed late in 1983. A straw-man implementation was developed in the summer and early fall of 1983 to test out the basic ideas. The straw-man used simplified design-rules and operated in only a single direction. In late 1983 we began implementation of a new version to handle real design rules and hierarchical designs. That implementation has just recently become operational, but it still works in only a single direction.

### **2.3. More Accurate Timing Models for Crystal (J. Ousterhout)**

The new slope-based timing models for the Crystal timing analyzer are now fully implemented and in regular use by designers. Their accuracy was benchmarked against the SPICE circuit simulator, using a dozen critical paths from

the RISCII processor and cache chips. On average, Crystal's delay estimates were within 5-10% of SPICE's estimates, even though Crystal's simplified delay calculation algorithm is approximately 10,000 times faster than SPICE's.

**References:**

- (1) J.K. Ousterhout, "Switch-level Delay Models for MOS VLSI," to appear, *21st Design Automation Conference*, June 1984.
- (2) J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "A Collection of Papers on Magic," Technical Report No. UCB/CSD 83/154, Computer Science Division, University of California, Berkeley, December 1983.
- (3) J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "Magic: A VLSI Layout System," to appear, *21st Design Automation Conference*, June 1984.
- (4) G.S. Taylor and J.K. Ousterhout, "Magic's Incremental Design-Rule Checker," to appear, *21st Design Automation Conference*, June 1984.
- (5) W.S. Scott and J.K. Ousterhout, "Plowing: Interactive Stretching and Compaction in Magic," to appear, *21st Design Automation Conference*, June 1984.

#### 2.4. Finite-State Machine Synthesis (R. Newton and A. Sangiovanni-Vincentelli)

Our research in this period has been concentrated on four major issues: finite-state machine synthesis, relaxation-based circuit simulation, special-purpose architectures for the solution of large scale systems, and simulated annealing techniques for placement and routing of standard cells, gate-arrays and macro-cells.

Sequential circuits play a major role in the control part of digital systems. We addressed the automated synthesis of sequential logic functions in a structured VLSI design methodology. We considered sequential logic functions implemented by synchronous deterministic Finite State Machines (FSM) consisting of two distinct components: a combinational circuit implemented in two levels of logic, using a Programmable Logic Array (PLA) or constrained gate-matrix style, and a memory implemented by Delay-type registers, such as a set of  $C^2MOS$  latches.

In particular we considered the problem of assigning binary codes to the internal states of a Finite State Machine. In our past research, we have proposed the generation of adjacency rules for the encoding of states based on heuristic techniques developed by extracting a few operations used by logic minimizers to reduce the number of product terms of a PLA realization of FSMs[1]. We also proposed new algorithms for the solution of the graph embedding problem arising from the rules determined by the heuristic rules. Unfortunately, most logic minimizers that have been developed recently are very complex programs based on the use of sophisticated techniques. Hence, the heuristic rules are only able to capture a small part of the operations of logic minimizers such as MINI, PRESTO, or the logic minimizer we developed some time ago, POP.

Recently, in collaboration with Dr. Brayton of the IBM T.J. Watson Research Center, we have been able to obtain a completely new approach to the state encoding problem. The new approach is based on the observation that the state table representation of a FSM can be thought of as a personality matrix of a PLA where the present states and the next states can be considered as symbolic or multiple-valued variables. Then, by using a symbolic or multiple-valued logic minimizer we can minimize the number of "product terms" (rows) in the state table representation of the FSM. This minimization provides a set of guidelines on how to encode states such that the minimization obtained by the symbolic minimizer can be achieved after the states have been encoded. In some sense, we have turned the problem on its side: minimization and state encoding are no longer two separate steps.

The "rules" obtained by applying the multiple-value logic minimizer tell us that a set of states should be assigned adjacent codes, i.e. they should be placed on the same hyper face of the Boolean hypercube representing the number of bits used in the encoding. Note that this approach defines a new combinatorial optimization problem which we conjecture to be NP-complete. We devised a heuristic algorithm with some guaranteed properties which assigns codes to states so that all the rules are satisfied. The dimension of the Boolean space may be larger than the minimum number of bits needed to encode all the states of the FSM, i.e., if  $n$  is the number of states, the ceiling of  $\log_2 n$ . An interesting open problem which we are investigating is the trade-off between the number of bits used in the encoding and the number of product terms after the encoding has been done. Note that to satisfy all the adjacency rules determined by the symbolic minimization we may need to use a large number of bits. However, such a large number of bits may be needed because of one or two rules. If we decide not to satisfy these rules, the number of product terms may be two or three larger but the number of bits used may be smaller, thus reducing the



overall area of the implemented PLA. The results of this research will be reported in a paper to be submitted to the ICCAD 1984.

In connection with this research, we have developed a new set of algorithms for logic minimization. These algorithms devised in collaboration with Dr. Brayton of IBM, Prof. Hachtel of the University of Colorado, Boulder and C. McMullen of Harvard University, represent a significant improvement over other available techniques. The resulting program developed under IBM sponsorship has resulted in an APL program and in a C program, ESPRESSO II, which is considerably faster than MINI but always generating better or equal results than MINI, PRESTO, or POP in the final implementation of the logic. In particular, we have been able to obtain a result that shows how to use a two-level minimizer to minimize multiple-value logic, thus making ESPRESSO II, born as a two level logic minimizer, an effective tool for FSM synthesis. We are developing a specialized set of algorithms for the multiple value logic minimization problem to speed up even further the symbolic minimization step attached to the FSM synthesis procedure. We are presently writing a monograph on the algorithms for ESPRESSO II, which should appear at the end of the summer.

## **2.5. Relaxation-based Circuit Simulation (A. Sangiovanni-Vincentelli, R. Newton)**

Over the past six years, a new class of algorithms, called Relaxation-Based Methods, has been applied to the electrical IC simulation problem. We have developed a number of simulators (RELAX and RELAX2, SPLICE1.6 and SPLICE2) that use different forms of these methods to provide as accurate, or more accurate, waveforms than standard circuit simulators such as SPICE2 or ASTAP with up to two orders of magnitude speed improvement for large circuits. These simulators have been used for the analysis of both digital and analog MOS ICs, and more recently for the analysis of Bipolar circuits. They use *relaxation* methods for the solution of the set of ordinary differential equations, (ODEs) which describe the circuit under analysis, rather than the direct, sparse-matrix methods on which standard circuit simulators are based.

During this period, we studied the numerical properties of the various methods for the analysis of MOS circuits and we presented them in a rigorous and unified framework in [2] and we improved our relaxation algorithms and their implementation in [3-5].

Recent results with the ITA algorithm are presented in [3]. In particular, we have used the program to analyze a number of large, industrial circuits with results as accurate as SPICE. The SPLICE1.7 program is now at approximately 100 sites and we are working actively with five of those sites.

In [4], we describe RELAX2.1, a new improved version of RELAX, a Waveform-Relaxation based simulator and the new algorithms implemented in the program. In particular, we were able to characterize the convergence behavior of the Waveform Relaxation Method on a class of circuits which required a large number of iterations to converge. The study of the convergence behavior has led into the concept of "windowing", i.e. of breaking up the time interval over which analysis has to be performed, in sub-intervals so that the algorithm applied in these sub-intervals exhibits fast convergence. In addition, techniques for the automatic partitioning of the circuit into subcircuit have been included in RELAX2, developed with the sponsorship of a grant from MICRO,

avoiding the tedious operation of manually entering the decomposition of the circuit into subcircuits. The decomposition into subcircuits can speed up any relaxation algorithm and we are planning to extend this technique to the SPLICE2 program.

SPLICE2 has been developed as an experimental "framework" for exploring both relaxation-based electrical simulation as well as mixed-level simulation. The program performs ITA-based relaxation electrical simulation, as well as iterated, relaxation-based switch and logic simulation. New results in the area of convergence criteria, including "waveform convergence", have been developed, as well as new cache-based event scheduling algorithms that are well-suited to the wide circuit time-constants present in a mixed-level environment[5]. Present work involves extensions to Register Transfer Level, a new form of analysis called ELogic (Electrical-Logic), and the inclusion of path analysis code for tagging critical circuit paths to be simulated in detail.

## **2.6. Special Purpose Architectures for the Solution of Large Scale Systems (A. Sangiovanni-Vincentelli, R. Newton)**

The solution of Large-scale Systems of both algebraic and differential Equations(LSE), is needed in the analysis and simulation of many engineering systems.

New architectures, in particular vector computers such as the CRAY 1, have inspired the design of new algorithms to exploit parallelism in the solution process. An important example is the program CLASSIE for the simulation of electronic circuits. Along these lines, peripheral array processors, such as the FPS164, can also be used in conjunction with hosts such as the VAX11/780 to speed up the solution process. However, this speedup is not enough to cope with the problems to be solved in the VLSI era.

The advent of VLSI technology has made the cost-effective design of special purpose machines possible. Examples of these machines are the Yorktown Simulation Engine(YSE) for logic simulation and the use of and Systolic Arrays. Special-purpose machines have also been proposed for the solution of linear, algebraic LSEs. Most of these machines limit the size of the operand matrix. When no size limit is imposed, the operand matrix has to be partitioned into submatrices of equal sizes. Only Johnsson and Pottle treated the related numerical properties. However special matrix structures, such as the Bordered Block Diagonal Form (BBDF) or the Bordered Block Triangular Form(BBTF), commonly expected in engineering problem, are not exploited in much of this work. In [6], we proposed a new algorithm-architecture BLOSSOM for the solution of LSE.

This architecture supports other matrix operations used as subprocedures by block LU decomposition such as the multiplication and the inversion of submatrices. We described the hardware implementation of these matrix operations. We are simulating the performance of the proposed architecture and comparing its speed and power with other architectures such as data-flow machines. In addition, we are studying the combinatorial optimization problem arising from the optimal partitioning of the sparse matrix. We are in the process of developing new algorithms for determining the optimal partition. In addition, we are investigating numerical techniques to guarantee the numerical stability of the scheme used in the special purpose hardware, a unique feature of BLOSSOM.

We have been investigating the use of data-driven computer architectures for use in circuit simulation since 1981. Early in 1982, we concluded that relaxation-based algorithms are well-suited to the use of data-driven multiprocessors for the solution process. Using the FTL2 program[7], we simulated a variety of computer interconnection networks while running a distributed version of the SPLICE ITA algorithm. Based on the results of those simulations, we decided that for our problem, the multi-stage perfect shuffle network, or OMEGA network, was the best choice. Unfortunately, we could not analyze large circuits on our simulated machine. We have since used the BBN Butterfly computer, developed at BBN for DARPA, as a vehicle for tuning our algorithms, analyzing their properties, and evaluating performance limits of the interconnection network. The Butterfly consists of a number of MC68000 processors, AMD2901-based processors, and memory, connected using an OMEGA net. The machine we used had 10 processing nodes but machines with up to 128 processors are planned at BBN.

Our results for the practical runs supported our earlier analysis. For the analysis of an industrial circuit containing over 700 MOSFETS we achieved 70% efficiency on the 10-processor machine. This compares well with the maximum 10-15% efficiency we have been able to achieve using direct analysis techniques on vector processors such as the CRAY 1. We are presently awaiting our own 16-processor Butterfly to continue this research and have developed a floating-point support board for the machine. We also plan to add additional memory to each node. We estimate that with 1MIP processors at each node, with 256 nodes in such a machine, we would be able to achieve routine 1000-fold performance improvement over SPICE2 on a single processor for the analysis of large circuits. The initial results of this research will be reported in [8].

## **2.7. Simulated Annealing Algorithms for Placement and Routing (A. Sangiovanni-Vincentelli, R. Newton)**

Simulated Annealing is a relatively new technique proposed by Kirkpatrick, Gelatt and Vecchi of the IBM T.J. Watson Research Center for the solution of complex combinatorial optimization problems. This technique has been introduced by establishing an analogy between combinatorial optimization and the annealing process. Algorithms based on this technique have been developed for the placement and routing problem. We believe that this technique has great potential for the solution of the placement problem for IC design. In particular, its simplicity and flexibility associated with its property of not being necessarily stuck at a local optimum (which is a common drawback of the heuristic used in layout problems) has attracted the attention of a number of researchers in the Universities as well as in Industry. We have developed a set of algorithms and corresponding packages for the placement of gate-arrays, standard cells and macro cells. We have recently tested the results obtained by the algorithms on a set of complex industrial circuits. In all cases, a significant reduction of chip area over the area obtained by other optimization techniques and the one obtained by hand-layout has been achieved[5]. The only drawback of this technique is long running time. In the largest example tried thus far, a 1,500 cell layout, a total reduction of chip area of the order of 37% has been obtained at the expenses of 24 hours of computing time on a Vax11/780 running VMS.

To the best of our knowledge, there is no theory available today to justify the good performance of the algorithm besides the physical analogy mentioned

above. We are presently trying to apply Markov chain theory to prove that in ideal situation the algorithm produces the *global* optimum with probability one. The results of the theoretical investigation should provide the necessary foundations of the method in addition to new vistas on methods to reduce the computing time.

#### References:

- (1) G. De Micheli, A. Sangiovanni-Vincentelli and T. Villa, " Computer-aided Synthesis of Finite-State Machines " , *Proc. of Int. Conf. on CAD 1983, Santa Clara, California*, Sept. 1983.
- (2) A. R. Newton and A. Sangiovanni-Vincentelli, " Relaxation-Based Electrical Simulation " , *IEEE Trans. on Elec. Dev.*, Sept. 1983, *SIAM Journ. on Scientific and Statistical Computing*, Sept. 1983,
- (3) R. Saleh, " Iterated Timing Analysis and SPLICE1, " ERL Memorandum Number UCB/ERL M84/2, Electronics Research Laboratory, University of California, Berkeley, 4 January 1984.
- (4) J. White and A. Sangiovanni-Vincentelli, " RELAX2.1: A Waveform-Relaxation Based Circuit Simulation Program " , to appear, *Proc. of Cust. Int. Circ. Conf.*, Rochester, May 1984.
- (5) J. Kleckner, " Techniques for Advanced Mixed-Mode Simulation, " Ph.D. dissertation, University of California, Berkeley, April 1984 (to be published as ERL Memo).
- (6) H. Ko and A. Sangiovanni-Vincentelli, " BLOSSOM: an Algorithm and Architecture for the Solution of Large-Scale Linear Systems " *Proceedings of the International Conference on Computer Design*, New York, Oct. 1983.
- (7) J. T. Deutsch and A. R. Newton, " Data-Flow Based Behavioral-Level Simulation and Synthesis " , *Proc. IEEE ICCAD Conference*, Santa Clara, Ca. Sept. 1983.
- (8) J. T. Deutsch and A. R. Newton, " A Multiprocessor-Based Implementation of Relaxation-Based Circuit Simulation, " to appear, *Proc. 21st Design Automation Conference*, Albuquerque, New Mexico, June 1984.
- (9) C. Sechen and A. Sangiovanni-Vincentelli, " TimberWolf: A Placement and Routing Package " , to appear, *Proc. of Cust. Int. Circ. Conf.*, Rochester, May 1984.



## 2.8. Tradeoffs in Wire Representation for VLSI Layouts (C.H. Sequin)

Most VLSI chips are, and will be even more so in the future, dominated by their interconnections. Correspondingly, the wiring task dominated the actual layout tasks in the RISC and SOAR chips. Effective tools for interactive or automatic wiring of VLSI chips are thus the most important missing link in our UNIX design environment. The basis of any tool that has to deal with interconnections is a suitable underlying semantic model and appropriate data structures for wires.

We have studied tile-based wire representations in the context of the space partitioning by *corner-stitched tiles* introduced by Ousterhout [1]. The representations studied can be divided into three broad classes:

*Skeletal representations* model a wire as a chain of connected line segments with attached attributes such as wire width and type of material.

*Purely physical representations* model the physical space occupied by wire material without considering the segment structure of the wire and without explicitly representing connectivity.

*Directed box representations* attempt to combine the best features of skeletal and purely physical representations by tiling the physical space occupied by the wire in ways that preserve a simple mapping between the tiles and the individual segments of the wire.

All representations have some pros and cons. Skeletal representations cleanly express connectivity but require more effort to identify potential interactions between objects. Purely physical representations fit most naturally into the framework of corner-stitched tiles but are less suitable for expressing and manipulating the underlying segment structure of the wire. The directed box representations turned out to be more limited than expected in their ability to express the ways in which wires connect to each other, to terminals, and through contacts to other levels.

An experimental program for incremental wire manipulation, WICRD [2], has been built to support this analysis and to study tradeoffs in the human interface to an IC routing tool. It explored in detail both the advantages and the ultimate limitations of the directed box wire representation used in WICRD. We have integrated a simple Lee maze router in WICRD and examined the routing problems that arise when wires are of varying width and when they can exceed the spacing of the wire-placement grid typically used for this algorithm. The first approach explored moved a larger square of wiring material through the layout while checking for violations of any constraints. An analysis of the space and time requirements of this algorithm then favored a second approach, in which the given (obstacle) geometry is suitably enlarged to reduce the problem of finding a routable path to the conventional problem for a minimum width wire. The WICRD system has also demonstrated that an incremental wire movement algorithm can be used effectively for global compaction.

### References:

- (1) J.K. Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools", *IEEE Trans. on Computer-Aided Design*, Vol. CAD-3, No. 1, Jan 1984, pp 87-100.
- (2) D. Wallace, "A Comparison of Tile-Based Wire Representations for Interactive IC Layout Tools", Master's Report (in preparation).

### **3. CIRCUIT & SYSTEM DESIGN**

#### **3.1. Design Frames and System Kits (R. H. Katz)**

A design frame is the hardware analog of a software operating system. It provides the necessary run-time support for providing system-level services to a user designed integrated circuit. It consists of a standard pad frame, interface circuitry, and printed circuit board to accept the chip footprint. We have been experimenting with a design frame for the MultiBus system bus.

A number of projects from the Fall offering of the Mead and Conway design course were designed for inclusion within the MultiBus design frame, and were submitted to MOSIS for fabrication in January. These include several simple implementations of the same 16-bit microprocessor architecture and a test configuration chip. A MultiBus design frame board has been designed and fabricated by the MOSIS service. At this time, we still await the return of the course project chips.

With the arrival of the printed circuit boards, we have just begun serious testing with a previously fabricated test circuit within the frame. We hope to be able to report on our successful integration of the frame chip and board within the SUN workstation by the time of the DARPA meeting.

A higher performance version of the design frame is currently being designed by a student (Gaetano Borriello) that corrects a number of the performance problems encountered during last semester's course. In addition, several projects are underway to build real systems within the design frame context. One student (Frankie Leung) is building a broadside integrated circuit tester controller around the design frame. A controller within the chip level frame will (1) download on-board test vector memory as a DMA device on the MultiBus, (2) cycle through the test vectors, and (3) upload the test results. Another student (Rick Brown) is building a scan logic controller as an extension of the configuration chip he designed last semester. This chip will provide a flexible means to read and write a scan path chained through a number of custom designed parts. The design frame is being extended with a scan logic subsystem for easier interfacing with the scan logic control chip.

A paper is currently in preparation describing the design frame concept, the course results, and future directions. The paper is being written in collaboration with Gaetano Borriello, Alan Bell of Xerox PARC, and Lynn Conway of DARPA.

#### **3.2. A 1000 Word Speech Recognition System (R.W. Brodersen)**

Circuitry contained on one multibus card has been developed which will be able to recognize 1000 words in real time. On the card are 2 custom I.C.'s, an Intel 80186 microcomputer and memory. This card has the capability of 116 MIPS of von Neumann equivalent instructions, thus demonstrating the power of dedicated parallel, pipelined processing.

One of the IC's is a filterbank chip which was generated fully automatically from software. This chip implements a 16 channel filterbank with a 112 poles of

filtering, at an initial sample rate of 14 kHz.

The 2nd chip is an enhanced version of a previous design, which performs a dynamic programming algorithm. The new chip has more parallelism as well as a number of glue logic functions for memory control which were required to be able to fit the entire recognition system on one board.

The 186 performs the multibus interface and will be used in future research to implement such things as syntax direction, continuous speech algorithms and sophisticated training (learning) algorithms. The flexibility of this board makes it an ideal candidate as a base for future speech recognition research.

We have put this card into a SUN (UNIX based) workstation, and plan to use it to incorporate speech into a number of applications. The UNIX drivers which allow direct interaction between the SUN cpu and the 80186 cpu on the board have been written. The software is now being developed which will allow a straight-forward high level interaction between application programs written on the SUN and the recognition board. Features such as adaptive training algorithms, syntax direction and multiple vocabularies will be supported.

Discussions are underway with several companies to produce and support the distribution of the board. In addition, SRI have generated a wirelist for the board to allow duplication of the board wiring by an automatic wire-wrap machine. It is expected that 10-20 boards will be made in this way for internal distribution at Berkeley and SRI.

#### References:

- (1) R. Kavalier, T. Noll, H. Murviet, M. Lowy and R.W. Brodersen, " A Dynamic Time Warp IC for a 1000 Word Recognition System " , *Proc. of ICASSP, San Diego, March, 1984*.

### 3.3. Real-time on-line handwriting recognition (R. W. Brodersen)

This project involves the design of an on-line handwriting recognition system to handle more than 500 custom symbols in real-time. The system consists of four major blocks: feature extraction, training, prematching, and postmatching. The algorithms for feature extraction, prematching and postmatching have been developed. Recent progress has been in the following areas:

- [1] The algorithms have been successfully implemented using the speech recognition hardware. Both accuracy and response time are satisfactory. We are now confident that handwriting recognition can be integrated with speech recognition in the same hardware without significant degradation in the character recognition accuracy.
- [2] The prematching algorithm has been modified so that it takes more advantage of the real-time processing capability of the speech processor hardware.
- [3] A clustering technique has been incorporated in the training, which allows a significant decrease in templates without a decrease in accuracy.

Future work will involve the implementation of the algorithms in the new speech recognition system and to write application programs which make use of the new form of interaction.

#### References:

- (1) Po-Yang Lu and R.W. Brodersen, "Real-time On-line Character Recognition", *To be published in the Pattern Recog. Conf., April, 1984*

### 3.4. Computer Generation of Digital Filter Banks (R.W. Brodersen)

A software, hardware system has been developed to automatically generate digital filter bank circuits from high level filter descriptions. This system now has been fully documented [1,2], and discussions are underway with several companies to give them access to the programs.

The software consists of two main programs: The filter compiler which converts the filter descriptions to hardware descriptions (micro code, RAM size, etc) and the layout generator which converts the hardware descriptions to a layout (mask) descriptions. To check the algorithms before the circuit is fabricated, a test set is used which runs the micro code on a data path the same as that included in the final circuit.

The major difference between this system and other automated design efforts is that this software and hardware was optimized for the specific groups of DSP applications of digital filter banks. This resulted in a software design time of only one month.

A number of circuits have been generated, fabricated and tested. A small single bandpass filter chip, a 18 channel spectrum analyzer for a speech recognizer which is being used in a speech recognition system and a 16 channel spectrum analyzer for consumer stereo have been fabricated. All circuits were functional the first time (the tester caught all errors) with a very short time required for filter specification and testing (approximately 1 day).

#### References:

- (1) P. Ruetz, S. P. Pope, B. Solberg and R.W. Brodersen, "Computer Generation of Digital Filter Banks", *ISSCC Digest of Technical Papers, Feb 1984*.
- (2) Ruetz, P. "Computer Generation of Digital Filter Banks", MS Project, Univ. of Calif., Berkeley, April, 1984.

### 3.5. Macrocells for Signal Processing (R.W. Brodersen)

The primary area of investigation is research in architectures and VLSI circuits suitable for signal processing applications.

A single-chip linear-predictive coding (L.P.C.) vocoder circuit has been designed and tested [1]. This is a device used for digitally encoding speech at a low bit-rate. Test results have shown that the device correctly performs the L.P.C. algorithm and functions reliably as a real-time, full duplex vocoder. Arrangements have been made with an industrial company experienced in speech coding to perform critical evaluations of the speech quality with the device configured into an LPC-10 coding system.

Another circuit which has recently been completed is a speech synthesizer which implements a recently developed multirate root LPC algorithm [2]. This algorithm uses a closed form analysis and a formant like structure. By using quadratic coefficients and section repeat its data rate can be lower than 1kbps. The quality can be continuously improved (with increasing bit rate) by including a representative residual.

Further work is being done in the area of signal processing I.C. design. Appropriate architectures and computer-assisted design techniques are being investigated. The use of multiprocessor architectures and macrocell-based circuit design techniques are being actively studied. Families of signal processing applications, such as speech processing and telecommunications, have been identified as suitable for implementation using these techniques. The goal of this research is to develop the tools and resources needed for rapid design of semi-custom I.C.'s for these applications.

### References

- (1) S. P. Pope, B. Solberg and R.W. Brodersen, " A Single-Chip LPC Vocoder " *Tech. Digest of the ISSCC*, Feb. 1984
- (2) C.C. Hsiao and R.W. Brodersen, " A Multirate Root LPC Synthesizer " , *Proceedings of ICASSP*, March, 1984.
- (3) C.C. Hsiao, " Multirate LPC Synthesis " , PhD Thesis, Dec. 1984, (to be published as an ERL Memo)



high for both  $\text{As}^+$  implanted and unimplanted specimens, indicating Ta is inadequate as a diffusion barrier at  $500^\circ$  for Al-Si contacts.

Current research work concentrates on the Ti-Si system and the application of rapid pulse lamp annealing to form refractory silicide-Si contacts.

and the later is due to Al spiking. Our test results shows that open failure is observed in both Al/n+ and Al/p+ contacts while leakage failure is observed only in Al/n+ contact. SEM examinations, however, show contact pits formation in both cases. A good explanation for that is Al contact to n-substrate forms a Schottky diode while Al contact to p-substrate creates a short. Therefore, no junction leakage increase is seen in Al/p+ contact.

The total contact chain resistance is also monitored. In Al/n+ case, such resistance decreases slightly initially and then increases until open-circuited. Such initially increase is due to barrier height reduction. In Al/p+ case, the total resistance remains constant initially and then starts increasing. In either case, once the total resistance starts increasing, the contact chain opens up in a very short period. SEM examinations show the open occurs on or near the contact closest to the anode, depending on metal compositions.

As low value of specific contact resistivity is desirable in VLSI technology, there is a concern on the lifetime of contact electromigration. The lower the specific contact resistivity, the higher the current density near the contact edge. Higher current density may cause contact fails in a shorter period. A study to investigate this trade-off is currently underway.

#### 4.10. Modification of Metal-Si Contacts With Ion Beams (N. Cheung)

We have completed the study of two metallization systems: the Al-Si and Ta-Si contacts. For the Al-Si system, a contact resistivity,  $\rho_c$ , of  $1.5 \times 10^{-6} \Omega\text{-cm}^2$  has been determined. After implantation of As (300 KeV) up to a dose of  $10^{15}/\text{cm}^2$  and a post-implantation annealing at  $450^\circ\text{C}$ , the substrate leakage current is substantially reduced. The effect can be attributed to the increased interface uniformity of the implanted contacts when compared to their unimplanted counterparts. The smoother interface morphology of the implanted interface has also been verified by scanning electron microscopy.

The contact resistivity of Ta-Si contacts is  $2.5 \times 10^{-6} \Omega\text{-cm}^2$  for the unimplanted samples after annealing at  $800^\circ\text{C}$ . The contact resistance,  $R_c$ , was found to be proportional to  $(\text{Area})^{1.4}$ . This power dependence is consistent with the current crowding effect under the leading edge of the contact, and it reduces the effective current carrying area. After As<sup>+</sup> implantation (400 KeV) to a dose of  $5 \times 10^{15}/\text{cm}^2$ ,  $\rho_c$  increases by an order of magnitude to  $20 \times 10^{-6} \Omega\text{-cm}^2$ . The substrate leakage current for both implanted and unimplanted contacts is much lower than that found for Al-Si contacts. For the implanted Ta-Si contacts, there is no indication of any contact pitting or uneven Ta/Si reaction. The increased contact uniformity is attributed to ion-beam mixing effects (i.e., the dispersion of interfacial contaminants and atomic mixing promote the uniform reaction of the Ta/Si interface).

A reliability study of Ta as a thin-film diffusion barrier has also been carried out. The samples were prepared by depositing 150Å of Ta over the entire wafer. The Ta film was then reacted with the substrate Si by As<sup>+</sup> implantation and post-annealing. The wafer was then subjected to a selective etch to remove the unreacted Ta over  $\text{SiO}_2$  regions. A thick layer of Al was evaporated and patterned using a lift-off process. The wafer was then annealed at  $500^\circ\text{C}$  for 60 minutes. Substrate leakage currents taken from  $2\mu\text{m}$  daisy chain patterns are

Albert Wu has made several specific contributions to this work. One is a computer program for automatically generating the layout rules from process bias information. He is also introducing new technology elements including a PRIST technique for protecting the poly during the p-channel source and drain implant.

#### **4.7. Simulation Aids for Viewing Topography from the Layout (A. Neureuther)**

A major milestone was accomplished in the completion of SIMPL-1 which is the CAD tool for Simulating Profiles from the layout using rectangular shapes. SIMPL-1 uses a file of process steps and the CIF layout information to automatically generate an SEM like view of the device topography. Any IC process sequence can be simulated by first describing the process steps from a menu of process elements. Examples of double N-well CMOS and Bipolar devices were given at the IEDM meeting. The documentation was completed in the MS report of Mike Grimm in December 1983 and a tape is being prepared for distribution. Work is being continued by Keunmyung Lee on data structures for a polygonal version and interface to other process simulators such as SAMPLE and SUPREM under SRC support.

#### **4.8. AL/SI Specific Contact Resistivity Measurement, (W.G. Oldham)**

We have demonstrated that, using a properly designed test pattern, specific contact resistivity can be consistently determined from contact end resistance using a transmission line model.

The results show that, for a given junction doping and a drive-in cycle, the specific contact resistivities are a constant regardless the sizes of contacts. Such measurement consistency and contact geometry independence have not been shown before by either conventional method or other methods published.

The dependence of specific contact resistivity on junction surface doping is also shown in agreement to that predicted by tunneling theory. Such agreement is good even when junction surface doping extends beyond commonly accepted electrically-active solid-solubility.

#### **4.9. AL/SI Contact Electromigration, (W.G. Oldham)**

The failures of Al/n+ and Al/p+ contact due to current stressing are being studied. In this study, an automatic measurement setup based on IBM Personal Computer is configured. In order to minimize temperature rise due to current heating, tests are conducted on test chips placed on a wafer hot chuck with a temperature controller.

There are two types of failures which can result from the stress, i.e. open circuit and junction leakage failures. The former is due to metal line opening

substrate resistance including the effects of guard rings. A new technique for suppressing latch-up was developed. A high energy (4MeV) implant of boron produces a buried p+ layer as a substitute for the widely used p on p+ epitaxial substrate. Since the high energy implant is performed after the long well diffusion, the

" epitaxial " layer thickness can be scaled more easily and is expected to be more uniform. Latch-up improvement is shown to match that of epitaxial substrates.

#### References:

- (1) K. Terrill, C. Hu, " Substrate Potential Calculation for Latch-up Modeling, " *accepted for publication IEEE Electron Device Letters*.
- (2) K. Terrill, C. Hu, " Prevention of CMOS Latch-up by High Energy Implantation, " *submitted to IEEE Electron Device Letters*.

#### 4.5. Soft Error Studies (C. Hu, A. Neureuther)

We published the computer analysis of the collection of alpha generated charge by collectors surrounded by either uniform reflecting or uniform absorbing surfaces. These are the two extreme cases of any real conditions that exists in IC. The analysis of the upper limit of charge collection should be more useful for circuit design than the previously available lower limit. The collected charge scales linearly with the linear dimension of the collector.

#### References:

- (1) K. Terrill, C. Hu, A. Neureuther, " Computer Analysis on the Collection of Alpha-Generated Charge for Reflecting and Absorbing Surface Conditions around the Collector, " *Solid State Electronics*, No.1 January 1984, pp. 45-52.

#### 4.6. Berkeley Advanced CMOS (A. Neureuther)

A collective effort is being made to bring up our advanced CMOS process in our new 4 " facility at Berkeley. Albert Wu under DARPA support is making a major contribution to this effort as the process coordinator. Several runs of the double N-well process on 2 " wafers have been completed by Kyle Terrill for latch-up studies and the device characteristics have been reasonable for channel lengths under 2  $\mu$ m. This process is being modified to create a base line process which is suitable for both the linear and digital needs of research groups in the Department, appropriate for our 4 " equipment and free of process bottlenecks such as occur with implants. This effort has been greatly aided by course work activities under the direction of professors Oldham and Neureuther. During the Fall Semester 20 students from the class designed test structures for device, process and yield on the 2x10 pad base for automatic testing. This semester 10 students are making the first run of 4 " wafers through the lab.

much too short a lifetime for operation at 7V or less.

Future work will fully clarify the mechanism for interface degradation and TDDB above and below 7V. (Alpha-particle, we now hypothesize, may induce oxide breakdown.) Methods for predicting oxide reliability will be developed.

#### References:

- (1) C. Hu, (Invited paper), " Charge Tunneling, Trapping, and Device Degradation in Thin SiO<sub>2</sub>, " *1983 IEEE Surfaces and Interfaces Specialists Conference*, Fort Lauderdale, Florida, December 1-3, 1983.
- (2) M.S. Liang, C. Chang, W. Yang, C. Hu, R.W. Brodersen, " Hot-Carriers Induced Degradation in Thin Gate Oxide MOSFET's " *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington D.C., December 1983, pp. 186-189.
- (3) C. Chang, M.S. Liang, C. Hu, R.W. Brodersen, " Carrier-Tunneling Related Phenomena in Thin Oxide MOSFET's, " *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington, D.C., December 1983, pp. 194-197.
- (4) S. Holland, I.C. Chen, T.P. Ma, C. Hu, " On Physical Models for Gate Oxide Breakdown, " *submitted to IEEE Electron Device Letters*

#### 4.3. Device Characterization and Modeling (C. Hu, P.K. Ko, R.S.Muller)

A well-equipped characterization laboratory has been set up. Besides hot-electron studies and thin oxide research described above, this laboratory is being used to study the charge pumping technique for MOS interface characterization. A simple quantitative model for the switch-induced error in switch capacitor circuits has been developed for the first time. A new MOSFET model suitable for CAD has been developed, and will be tested in a special version of SPICE. Research is underway to model and characterize the lightly-doped-drain (LDD) transistors.

#### References:

- (1) B.J. Sheu, C. Hu, " Modeling the Switch-Induced Error Voltage on a Switched Capacitor, " *IEEE Transactions on Circuits and Systems*, CAS-30 No.12, December 1983, pp.911-913.
- (2) B.J. Sheu, C. Hu, " Switch-Induced Error Voltage on a Switched Capacitor, " *accepted for publication in IEEE Journal of Solid State Circuits*.
- (3) B.J. Sheu, D.L. Scharfetter, C. Hu, D. O. Pederson, " A Compact IGFET Charge Model, " *submitted to IEEE Circuits and Systems Letters*.

#### 4.4. CMOS Latch-up Studies (C. Hu)

While qualitative models for CMOS latch-up abound, there is little research to quantify the problem. We have developed a model for the (distributed)

## 4. TECHNOLOGY

### 4.1. Hot Electron Studies (C. Hu, P.K. Ko, R.S. Muller)

A comprehensive review of the results of our research under the sponsorship of DARPA was presented at the 1983 IEDM in an invited paper titled " Hot-Electron Effects in MOSFET's ". A special effort was made to publicize our models in terms that are easily understood by IC engineers. A quantitative theory of photon generation by hot electrons with excellent agreement with experiment was completed and has been accepted for publication. Photon emission was also observed from forward-biased pn junctions. An in-depth study of our lucky-electron model for channel hot electron emission was written up and has been accepted for publication. We have observed significant channel-width dependence of hot-electron effects and are preparing a report on it. An MS and a Ph.D. thesis were completed during this report period.

Future research will determine the mechanisms responsible for MOSFET degradations and search for a reliable method to forecast device lifetime and screen unreliable devices. We will also study new device structures that minimize the hot electron effects.

#### References

- (1) C. Hu (invited paper), " Hot-Electron Effects in MOSFET's ", *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington, D.C., December 1983, pp. 176-181.
- (2) S. Tam, C. Hu, " Hot Electron Induced Photo-Carrier Generation in Silicon MOSFET's ", *accepted for publication in IEEE Transactions on Electron Devices*.
- (3) T.C. Ong, K. Y. Terrill, S. Tam, C. Hu, " Photo Generation in Forward-biased Silicon PN Junctions, " *IEEE Electron Device Letters*, EDL-4, No.12, December 1983, pp. 460-462.
- (4) S. Tam, P.K. Ko, C. Hu, " Lucky-Electron Model of Channel Hot Electron Injection in MOSFET's ", *accepted for publication in IEEE Transactions in Electron Devices*.

### 4.2. Thin Oxide Studies (C. Hu, R.W. Brodersen)

We were invited to review the results of our DARPA-sponsored research on the charge, transport, trapping, and degradations in thin oxides at the 14th Semiconductor Interface Specialists Conference. Two papers were presented at the 1983 IEDM. One addressed the physical nature of hot-carrier induced damage in MOS systems and experimental proof that hot holes play a negligible role in the process. The other presented the first direct measurement of the quantum yield of impact ionization as a function of electron energy among other results. A full paper on the subject is being prepared. As predicted in the last semiannual report, much progress was made on the study of the time-dependent dielectric breakdown of oxides in this period. A survey was presented at the 1983 Wafer Reliability Assessment Workshop. One completed paper reports clear evidence that holes generated by impact ionization are responsible for TDDB. Other known models such as one based on electron trapping are shown to be incorrect. We also found the standard method of accelerated testing to predict



PUBLICATIONS ON ARCHITECTURE

## ARCHITECTURE

The following section contains papers and reports relating to research in computer Architecture. They describe work which was wholly or in part performed under the sponsorship of the DARPA grant.

- (1) R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, and C.H. Séquin, "A 32b NMOS Microprocessor with a Large Register File", *ISSCC 1984, Digest of Tech. Papers*, San Francisco, Feb. 1984, pp. 168-169.
- (2) M.G.H. Katevenis, "Reduced Instruction Set Computer Architecture for VLSI", Ph.D thesis, U.C. Berkeley, Oct. 1983.
- (3) R.W. Sherburne, "Processor Design Tradeoffs in VLSI", Ph.D thesis, U.C. Berkeley, (~April, 1984).
- (4) R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, and C.H. Séquin, "Local Memory in RISCs", *Proc. Intl. Conf. on Computer Design*, New York, Oct. 31-Nov. 3, 1983, pp 149-152.
- (5) Y. Tamir and C.H. Séquin, "Strategies for Managing the Register File in RISC", *IEEE Trans. on Computers*, C-23, No. 11, Nov. 1983, pp 977-989.
- (6) W-H Yu, "LU Decomposition on a Multiprocessing System with Communication Delay", Ph.D. Thesis, University of California, Berkeley, CA, March 1984.
- (7) H-H Lu, "High Speed Recursive Filtering", Ph.D. Thesis, University of California, Berkeley, CA, Dec. 1983.
- (8) R.M. Fujimoto, "VLSI Communication Components for Multicomputer Networks," Ph.D Thesis, U.C. Berkeley, Fall 1983.
- (9) R.M. Fujimoto, "SIMON, A Simulator of Multicomputer Networks," Tech. Report No. UCB/CSD 83/140, Sept. 1983.
- (10) Y. Tamir and C.H. Séquin, "Self-checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *Proc. Intl. Conf. on Computer Design*, New York, Oct. 31-Nov. 3, 1983, pp 561-564.
- (11) Y. Tamir and C.H. Séquin, "Design and Application of Self-Testing Comparators Implemented with MOS PLAs," scheduled for publication in *IEEE Transaction on Computers*, Spring, 1984.

## SESSION XII: MICROPROCESSORS AND MICROCONTROLLERS

Chairman: Dana Secombe

Hewlett-Packard Co

Ft. Collins, CO

## THAM 12.1: A 32b NMOS Microprocessor with a Large Register File\*

Robert W. Sherburne, Jr., Manolis G.H. Katevenis, David A. Patterson, Carlo H. Sequin

University of California

Berkeley, CA

TWO SCALED VERSIONS of a 32b Reduced Instruction Set Computer<sup>1</sup> CPU - RISC II - have been implemented in a 5-mask NMOS process using layout rules<sup>2</sup> with Lambda-values of 2 $\mu$ m and 1.5 $\mu$ m (corresponding to drawn gate lengths of 4 $\mu$ m and 3 $\mu$ m, respectively). This approach has resulted in two surprisingly powerful processors.

The RISC II architecture uses a small, but carefully selected set of simple instructions that support the most frequently occurring operations. It relies on an orthogonally partitioned 32b instruction format and regular execution timing, affording a reduction of the amount of control circuitry required on chip. While most current microprocessors use from 40% to 70% of the chip area for control, in RISC II the control section occupies less than 10% of the chip. The regular instruction execution also permits the realization of a simple, yet efficient pipeline. In RISC II, the machine cycle time is limited only by the speed of the datapath, rather than by a long control path through a large microstore.

The area freed up by the simplification of the control logic has been used to add a significant amount of local memory in the form of a large register file, organized as ten global registers plus eight bank of 22 local registers each, with an overlap of 6 registers between adjacent banks. These register banks are used to support procedures: the overlap registers are used for parameter passing, and the others for local scalar variables. In this manner, the most frequently used operands reside on chip, and the data memory traffic, which often constitutes a serious limitation of overall system performance, is significantly reduced.

The CPU chip photomicrograph is shown in Figure 1. Chip area is dominated by the highly regular datapath. More than half of its length is occupied by the register file with a total of 138 32b registers. The control circuitry lies in the top right portion of the chip. It consists of a few small PLAs, rather than a microcode ROM. The actual opcode decoder (0.5% of chip area) is a generalized decoder consisting of an AND plane and a single row of OR gates, which makes it smaller and faster than a full PLA.

The organization of the three-stage pipelined datapath is shown in Figure 2. It is based on a 2-bus, dual-port static register file. The first stage of pipelining includes instruction

fetch and decoding. In the second stage, two operands are read from the register file and an ALU or shift operation is performed. The result is written back to the register file during the third stage; page-fault and other interrupts and traps are handled simply by aborting this last stage of the pipeline, as it is the only one that modifies user-visible state. The ALU or shifter result may be forwarded directly to the next executing instruction before it is written back into the register file. Thus data dependency delays are eliminated.

On branch instructions, the pipeline is not flushed; transfer of control is simply delayed by one pipeline stage. This delay is accommodated in software by defining all branch instructions to take effect only after the subsequent instruction. An optimization pass of the compiler can often put a useful instruction into this slot after each delayed branch instruction.

For accessing data memory external to the chip, normal instruction fetch and execution is suspended between the second and third pipeline stages to make the single I/O bus available for data traffic. The timing within each four-phase cycle allows up to three phases of the cycle for accessing external memory.

Dynamically precharged busses are used wherever possible. The ALU uses a dynamic carry chain with buffering every four bits. A single, 32b crossbar shifter with precharged, bidirectional busses is included for both right and left shifts.

Register file operation is illustrated with Figure 3. All bitlines are initially precharged during  $\phi_4$ . Simultaneously, the word lines are clamped to ground, and address decoding takes place. The grounded depletion transistor reduces the delay in the NOR decoder by isolating the input parasitics. The full 5V signal is delivered to the wordline driver by the clocked depletion mode pass transistor. The bootstrapped wordline driver is clocked during  $\phi_1$ . The selected bit cell then may discharge one of its associated bitlines. A single-ended sensing scheme was chosen due to its simple and compact realization. Register write is performed in the usual manner by differentially driving the bitline pair; the decoder are shared for both read and write addresses.

The two designs were implemented through silicon foundries<sup>3</sup>. Both chips were fully functional on first silicon. Design verification was made at the layout level (design rule, electrical rule, and label checking), the circuit level (switch level simulation, delay path analysis, and SPICE simulation of critical paths), and at the register transfer level (SLANG). The larger chip, for which all the simulation and delay analysis was carried out, ran within 5% of expected performance. Its instruction cycle time was 500ns rather than 480ns, corresponding to an 8MHz clock. The scaled down chip, for which no additional simulation had been done, performed with a machine cycle of 330ns (12MHz clock rate). Chip specifications are summarized in Table 1.

\*Research sponsored, in part, by Defense Advance Research Projects Agency (DoD) ARPA Order No. 3803, Monitored by Navat Electronic System Command under Contract No. N00039-81-K-0251.

<sup>1</sup>Patterson, D.A. and Sequin, C.H., "A VLSI RISC", IEEE Computer, Vol. 15, No. 9, p. 8-12; Sept., 1982.

<sup>2</sup>Mead-Conway.

<sup>3</sup>Xerox Palo Alto Research Center and DARPA-Sponsored MOS-Implementation Service (MOSIS) at the Information Science Institute.

Overall, the RISC approach has resulted in an architecture that support the most frequently occurring operations in an effective manner. The high computational throughput is due to the good match of the RISC architecture with the needs and constraints of VLSI single-chip processors.

#### Acknowledgment

The authors wish to thank R. Campbell for his efforts in performance evaluation.

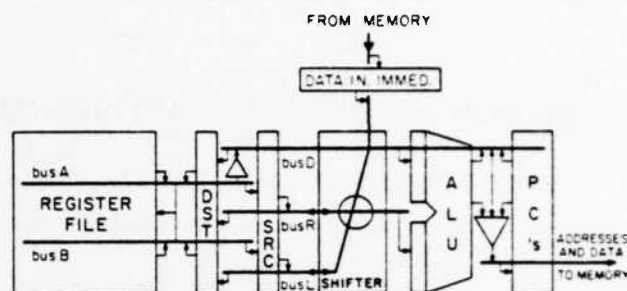


FIGURE 2—Datapath organization.

SPECIFICATIONS	CHIP A	CHIP B
DRAWN GATE LENGTH	4 $\mu\text{m}$	3 $\mu\text{m}$
DIE SIZE ( $\text{mil}^2$ )	228 x 406	171 x 304
REGISTER CELL AREA	4.6 $\text{mil}^2$	4.6 $\text{mil}^2$
CLOCK RATE	8 MHz	12 MHz
POWER DISSIPATION	1.25 W	1.83 W
REGISTER FILE	138 x 32 bits	
TRANSISTOR COUNT	40706	
PIN COUNT	62	
DESIGN TIME	2.8 man-years	

TABLE I—Chip specifications.

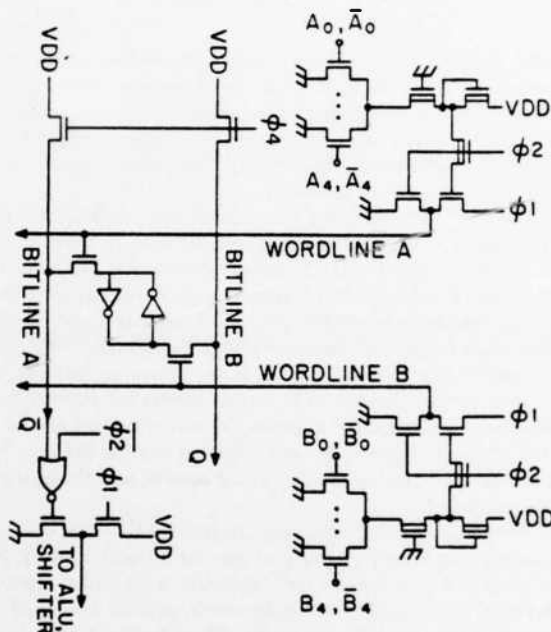


FIGURE 3—Dual-port register file read circuitry.

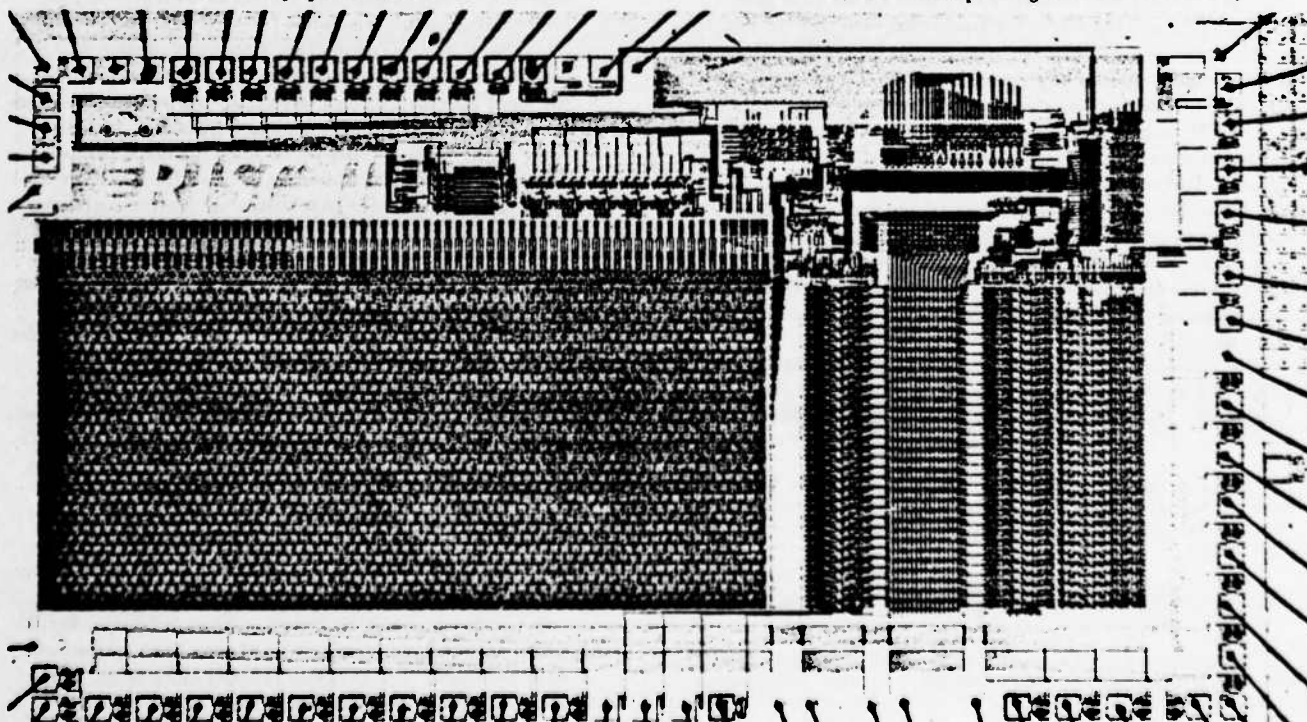


FIGURE 1—CPU chip photomicrograph.

REDUCED INSTRUCTION SET  
COMPUTER ARCHITECTURES FOR VLSI.

*Manolis G.H. Katevenis*

*Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley, Ca 94720.*

*Doctoral Dissertation*

*October 1983.*

## REDUCED INSTRUCTION SET COMPUTER ARCHITECTURES FOR VLSI

*Emmanuel-Manolis George Katsenivis*

### ABSTRACT

Integrated circuits offer compact and low-cost implementation of digital systems, and provide performance gains through their high-bandwidth on-chip communication. When this technology is used to build a general-purpose von Neumann processor, it is desirable to integrate as much functionality as possible on a single chip, so as to minimize off-chip communication. Even in Very Large Scale Integrated (VLSI) circuits, however, the transistors available on the limited chip area constitute a scarce resource when used for the implementation of a complete processor or even computer, and thus, they have to be used effectively. This dissertation shows that the recent trend in computer architecture towards instruction sets of increasing complexity leads to inefficient use of those scarce resources. We investigate the alternative of Reduced Instruction Set Computer (RISC) architectures which allow effective use of on-chip transistors in functional units that provide fast access to frequently used operands and instructions.

In this dissertation, the nature of general-purpose computations is studied, showing the simplicity of the operations usually performed and the high frequency of operand accesses, many of which are made to the few local scalar variables of procedures. The architecture of the RISC I and II processors is presented. They feature simple instructions and a large multi-window register file, whose overlapping windows are used for holding the arguments and local scalar variables of the most recently activated procedures. In the framework of



the RISC project, which has been a large team effort at U. C. Berkeley for more than three years, a RISC II nMOS single-chip processor was implemented, in collaboration with R. Sherburne. Its microarchitecture is described and evaluated, followed by a discussion of the debugging and testing methods used. Future VLSI technology will allow the integration of larger systems on a single chip. The effective utilization of the additional transistors is considered, and it is proposed that they should be used in implementing specially organized instruction fetch-and-sequence units and data caches.

The architectural study and evaluation of RISC II, as well as its design, layout, and testing after fabrication, have shown the viability and the advantages of the RISC approach. The RISC II single-chip processor looks different from other popular commercial processors: it has less total transistors, it spends only 10% of the chip area for control rather than one half to two thirds, and it required about five times less design and layout effort to get chips that work correctly and at speed on first silicon. And, on top of all that, RISC II executes integer, high level language programs significantly faster than these other processors made in similar technologies.

Carlo H. Séquin

C. H. Séquin (Committee Chairman)

10/10/1983

Αφιερώνεται  
στους Γονείς μου,  
που μ' έμαθαν να χρησιμοποιώ  
λογική και τάξη στο να λύνω τα προβλήματα,  
κι αγάπη για τον Άνθρωπο στο να τα διαλέγω.

*Dedicated  
to my Parents,  
who taught me to use  
logic and order for solving problems,  
and love for People in choosing problems to work on.*

## Table of Contents

### Chapter 1:

<b>INTRODUCTION.</b> .....	1
<b>1.1 The RISC Concept:</b>	
<b>Effective Use of Scarce Hardware Resources.</b> .....	1
<b>1.2 Evolution of the Berkeley RISC Project.</b> .....	6
<b>1.3 Thesis Organization.</b> .....	8
<b>References.</b> .....	9

### Chapter 2:

#### THE NATURE

<b>OF GENERAL-PURPOSE COMPUTATIONS.</b> .....	11
<b>2.1 Goal and Methods of Program Measurements.</b> .....	12
<b>2.2 Review of some Program Measurements from the Literature.</b> .....	16
<b>2.3 Study of some Critical FORTRAN Loops</b>	
<b>(collected mostly by Knuth).</b> .....	21
<b>2.4 A Study of four C Programs</b>	
<b>for Text Processing and CAD of IC's.</b> .....	28
<b>2.5 Summary of Findings.</b> .....	40
<b>References.</b> .....	41

**Chapter 3:**

<b>THE RISC I &amp; II ARCHITECTURE AND PIPELINE .....</b>	<b>42</b>
3.1 The RISC I & II Instruction Set. ....	43
3.2 The RISC I & II Register File	
with Multiple Overlapping Fixed-Size Windows. ....	55
3.3 The RISC I & II Pipelines. ....	67
3.4 Evaluation of the RISC I & II Instruction Set. ....	77
References. ....	89

**Chapter 4:**

<b>THE RISC II DESIGN AND LAYOUT. ....</b>	<b>92</b>
4.1 The RISC II Data-Path,	
and its Use for Instruction Execution. ....	92
4.2 The RISC II Timing. ....	102
4.3 The RISC II Control. ....	113
4.4 Design Metrics of RISC II. ....	120
References. ....	122

**Chapter 5:**

<b>DEBUGGING AND TESTING RISC II. ....</b>	<b>123</b>
5.1 Logic Debugging Tools and Methods. ....	124
5.2 Testing the RISC II Chips. ....	130
References. ....	136

**Chapter 6:****ADDITIONAL HARDWARE SUPPORT****FOR GENERAL-PURPOSE COMPUTATIONS. .... 138****6.1 Multi-Window Register Files versus Cache Memories****for Scalar Variables. .... 140****6.2 Fixed-Size, Variable-Size, and Dribble-Back****Multi-Window Register Files. .... 145****6.3 Support for Fast Instruction Fetching and Sequencing. .... 154****6.4 Pointers and Data Caches. .... 170****6.5 Multi-Port Memory Organization. .... 177****References. .... 183****Chapter 7:****CONCLUSIONS. .... 185****Appendix A:****DETAILED DESCRIPTION****OF THE RISC II ARCHITECTURE. .... 189****A.1 User-Visible State of the CPU. .... 190****A.2 Interface between CPU and Outside World. .... 194****A.3 Instruction Execution Sequencing. .... 199****A.4 Instruction Set. .... 200****A.5 Interrupts and Traps. .... 210**



## ACKNOWLEDGMENTS

It was only through a coordinated team effort that the Berkeley RISC project evolved from a concept to the reality of silicon chips and a C compiler. I would like to thank all those individuals who contributed to this effort, some of whom are mentioned in § 1.2. In particular I would like to thank three of them. Carlo Séquin, my advisor and committee chairman, greatly helped me in all aspects of my doctoral research, as well as in improving my technical writing skills. Dave Patterson was the main driving force of the RISC project, judiciously guiding the choice of the reduced instruction set and many design decisions, organizing the initial 4-course sequence where the original study of the RISC concept was carried out, and successfully coordinating the many people and groups that were working on RISC. Bob Sherburne is the person with whom I spent long days designing, laying-out, debugging, and testing the RISC II chip; it is with his deep knowledge of IC design, and with his dedicated collaboration that we were able to make the RISC II chip a working high-performance reality.

There are several other people that I would like to thank a lot. Kjell Doksum helped me with useful comments on this thesis. John Ousterhout created, maintained, and revised *Caesar*, our principal design tool. Lloyd Dickman originally suggested the use of a three-stage pipeline. Dan Fitzpatrick and John Foderaro helped with design tools and with software problems.

In this research I was supported for two years by an IBM graduate student fellowship. The RISC project was supported in part by ARPA Order No. 3803, and monitored by NESC #N00039-78-G-0013-0004.

## CHAPTER 1:

# INTRODUCTION.

Even in Very Large Scale Integrated (VLSI) circuits, the number of transistors available on a single chip must be considered a limited resource. In the course of the "Reduced Instruction Set Computer" (RISC) project at U. C. Berkeley, it was found that hardware support for complex instructions is not the most effective way of utilizing the transistors in a VLSI processor. In chapter 1, the RISC concept is presented first, followed by an overview of the Berkeley RISC project, and some notes on the organization of this thesis.

### 1.1                      The RISC Concept: Effective Use of Scarce Hardware Resources.

Increasing the size or complexity of a digital circuit may either enhance or deteriorate the overall system performance, depending on how judiciously the added complexity is chosen. The Berkeley RISC project has demonstrated the viability of general purpose computers with simple instruction sets. It has brought concrete evidence showing the non-optimal utilization of silicon

resources in most contemporary single-chip processors, due to the increased complexity of their instruction sets.

### 1.1.1 Size, Complexity, and Speed.

Increasing the size or complexity of a digital circuit may lead to better system performance. For example:

- A 32-bit adder will allow a 32-bit processor to operate at higher speed than a 16-bit adder, used twice for each addition, would allow it to operate.
- Overlapping instruction execution with the fetching of the next instruction reduces the execution time of programs with an average number of jump instructions.
- Including, for example, 4 registers into a general-purpose CPU will give a much better performance than if only 2 registers were included, -- if the compiler can take advantage of additional registers.

All these are examples of cases where the increase in size or complexity is used to allow parallel execution of common parallel operations, or to provide faster access to frequently used operands.

On the other hand, increasing the size or complexity of a digital circuit can also have negative effects on its performance:

- A larger size entails longer wire delays.
- More gates mean less power is available per gate, resulting in reduced driving strength.
- A more complex mode of operation usually means interposing more circuit elements in the path of information flow: for example, additional or larger input multiplexors, increased output fanout, or more circuits hanging off busses. This inevitably reduces the maximum possible operating speed.

In VLSI systems this trade-off between speed and size/complexity of a circuit is more pronounced than it is in the previous-generation systems built from TTL SSI/MSI parts. The following tables show typical capacitances and delay

times in the TTL technology and in the NMOS process by which RISC II was fabricated:

Typical Delays of TTL inverters (7404) or 3-state buffers (74240):		
	S-series	LS-series
$C_L = 15pF$	3 ns	10 ns
$C_L = 50pF$	5 ns	15 ns

Typical Delays of $4\mu m$ NMOS inverters or buffers:		
	high-power	low-power
$C_L = 0.1pF$	3 ns	10 ns
$C_L = 2.5pF$	15 ns	60 ns

In VLSI MOS technologies, the gate delays vary over much wider ranges than in discrete technologies. The reason is twofold. On the one hand, the load capacitance significantly influences the delay time of MOS gates, while for discrete parts a large portion of the delay is due to the internal circuitry and to the package and does not depend so strongly on  $C_L$ . On the other hand, custom MOS offers much wider design choices in terms of size of devices, type of circuits available, and size of load to be driven. Thus, the dependence of system speed on size and complexity is much more direct in VLSI than it is in older, discrete technologies.

Another important factor in VLSI system design is the large difference in available bandwidth between on-chip and off-chip communication. In today's (1983) technology, one may typically see transfers on the order of 200 bits every 20 ns on-chip, versus only 50 bits going through the chip periphery every 50 ns. This character of the chip periphery as communications bottleneck makes it desirable to pack as much functionality as possible into the restricted area of a single chip. In this context, an increase of the size and complexity of one circuit feature may only be achieved at the expense of another.

Taking these trade-offs between size/complexity and speed properly into account, leads to hierarchically organized systems, where the inner units are physically smaller and support the most frequent operations. The system's architect has the important role of selecting the functions to be supported at the various levels of the system's hierarchy. This is particularly important in VLSI system design, where the spectrum of possible choices is wider and more continuous than it is in systems employing discrete technology.

### 1.1.2 Recent Trends, and the RISC Alternative.

A general trend in computers today is to increase the complexity of architectures commensurate with the increasing potential of implementation technologies, as exemplified by the complex successors of simpler machines. Compare, for example, the DEC VAX-11 to the PDP-11 [Stre78], the IBM System/38 to the System/3 [Utle78], and the Intel iAPX-432 to the 8086 [Tyne81] [Orga82].

Following the discussion made in the previous subsection, it is necessary to study the overall effect of such complex instruction sets on performance. For a VLSI system, does this approach lead to an effective utilization of the scarce silicon resources? In 1980, the "Reduced Instruction Set Computer" (RISC) project was started at U. C. Berkeley, with the goal of investigating an alternative to this trend. The hypothesis was that, since complex instructions are rarely used by actual programs, their inclusion into the processor's instruction set has more negative effects on overall performance than it has positive ones. On the other hand, the frequent program accesses to operands justify better support than is normally available in traditional architectures. A third consideration was that a simplified architecture is important in a field of such a rapidly changing technology, because it leads to a short design and debugging time, thus allowing quick exploitation of the new technologies.



From these considerations the Berkeley RISC Architecture was derived. It specified a general purpose processor with simple instructions and with many registers organized in multiple register banks. The RISC project has now (1983) demonstrated not only the viability but also the very definite advantages of this approach. The judicious choice of the instruction set was a key to this success. First, the most necessary and frequent operations (instructions) in programs were identified. Then, the data-path and timing required for their execution was identified. And last, other *frequent* operations (instructions), which *could also fit* into that data-path and timing, were included into the instruction set.

During the definition of the RISC architecture, its implementation was kept in mind at all times. The resulting architecture lies on a "knee of the curve" of the speed-versus-complexity trade-off. A significant number of commonly-used instructions is included in the ISP description of RISC (§ 3.1); all of them are implementable with a simple data-path and timing scheme. Including more instructions into the ISP would have required significant changes to the hardware, thus slowing down the cycle time.

U. C. Berkeley is not the only place where research on simple instruction sets is going on. Similar investigations are being carried out at IBM Watson Research Center in the 801 project [Radi82], and at Stanford University in the MIPS project [Henn82] [Henn83].

## 1.2 Evolution of the Berkeley RISC Project.

Table 1.2.1 shows the key steps in the history of the Berkeley RISC project. Two faculty members and about two dozen graduate students have been involved in this three-year project. The author of this dissertation has been heavily involved in it, starting with the architectural studies in the spring of 1980; subsequently his main concern was focused on the definitions of the micro-architectures for both NMOS versions and on design, layout, and debugging of RISC II.

<i>Table 1.2.1: History of the RISC Project.</i>		
Period	Activity	People
wint.80	RISC idea	Patterson, Séquin
spr.80	Architectural Studies	Patterson, 15 grad. stud.
sumr.80	Architecture Definition	Patterson, 4 grad. stud.
sumr.-fall.80	Compiler, Assem., Simul.	Campbell, Tamir
sumr.-fall.80	RISC I Micro-Architecture	Katevenis
wint.81	RISC II Micro-Architecture	Katevenis
wint.-spr.81	RISC I Design & Layout	Fitzpatrick, Foderaro, Peek, Peshkess, VanDyke
sumr.81-spr.82	RISC I fabrication	MOSIS, XEROX
sumr.82	RISC I tested	Foderaro, VanDyke
spr.-sumr.82	RISC I board	VanDyke
wint.81-wint.83	RISC II Design & Layout	Katevenis, Sherburne
spr.83	RISC II fabrication	MOSIS, XEROX
sumr.83	RISC II tested	Katevenis, Sherburne
1981-82	RISC/E ECL Paper Design	Beck, Davis, et.al.
spr.-fall.82	I-cache Design & Layout	Hill, Lioupis, Nyberg, Sippel
spr.83	I-cache fabrication	MOSIS, XEROX
sumr.83	I-cache tested	Lioupis, Hill
fall.82-fall.83	CMOS RISC Layout Study	Takada
wint.83-ongoing	RISC II microcomputer	Lioupis, Campbell

The Berkeley RISC architecture was defined in 1980, after extensive architectural studies performed during a graduate course. These included the measurement of several program parameters, such as the number of various statements and addressing modes, usage of local scalars, and procedure nesting depth. The measurements were done mostly in C, and also in Pascal, and did

not include any numeric computations program. This is the applications area for which the RISC architecture was designed. The author of this dissertation contributed to the above studies with a preliminary look at the data-path and the timing for such an architecture. Once the architectural design was finalized, he defined a micro-architecture to implement it. This was described in detail in [Kate80], and was subsequently adopted by a group of 5 graduate students who designed, laid-out, and debugged the corresponding NMOS IC in only six months. It was originally called "RISC I Gold", and later on simply "RISC I". Its very short design time was due to the simplicity of the architecture [Fitz81]. It was fabricated and tested; the chips were functionally correct, but slower than intended by about a factor of 4 [FoVP82]. This was due to a lack of tools, at that time, that could find all the critical timing paths in a simulation of the whole chip. A RISC I board, with memory and I/O around the CPU chip, was built by VanDyke and used to demonstrate the execution of small programs.

In parallel with the design of RISC I, the present author defined a second, more ambitious micro-architecture for the same processor architecture, and subsequently implemented it, together with Robert Sherburne, by designing, laying-out, and debugging a second NMOS IC. This was originally called "RISC I Blue", and later on "RISC II". It was fabricated and tested in 1983. The chips are functionally correct and work very close to predicted speed. RISC II occupies 25 % less silicon area than RISC I, even though it has 75 % more registers. This was made possible by reducing the number of busses that go through the register file from three to two, which led to a much more compact register cell. To avoid a resulting performance loss, an additional pipeline stage was used, as suggested by Lloyd Dickman. Overall, the circuit design and layout for RISC II was done with careful attention to performance.

Other parts of the RISC project have been going on in parallel: A detailed paper design was made for RISC/E, a RISC CPU and cache memory made out of SSI/MSI ECL IC's [Blom83]. Another group designed and laid-out an Instruction-Cache chip for the RISC II CPU [Patt83]. Cache chips were fabricated and tested; they were found to be functionally correct and to work very close to the predicted speed. A CMOS version of the RISC II micro-architecture was studied by M. Takada by designing and laying-out the data-path and most of the control. Finally, there is ongoing work, by Lioupis originally [Liou83] and by Campbell now, for designing and building a micro-computer around the RISC II CPU and I-cache chips.

### 1.3 Thesis Organization.

Since it is crucial for an architect to know which are the most frequently used operations (instructions), chapter 2 reviews the relevant literature on program measurements and complements it by a study of program properties done with a different method, providing yet another point of view.

The next three chapters deal with the architecture, micro-architecture, design, layout, debugging, and testing of RISC II. They show how the Berkeley RISC architecture fits into the concept of effective utilization of the hardware resources, and they present the most important experiences gained and conclusions reached from the whole cycle of micro-architecture definition to design, layout, debugging, and testing. It is appropriate here to make a clarification as to the terminology used. The term "RISC architecture" is general and refers to any architecture inspired by the "RISC concept" as presented in section 1.1.

The term "the Berkeley RISC architecture" refers to the specific RISC architecture defined at U.C.Berkeley in 1980 and implemented by RISC I and RISC II; sometimes we may abusively use the shorter term: "the RISC architecture".

The thesis concludes with a projection into the future. Soon, VLSI chips will have significantly more transistors than were used by RISC I or RISC II. What will these additional transistor be used for? Chapter 6 proposes additional hardware organizations, always within the framework of simple instruction sets, which will make effective use of those transistors for speeding up the execution of general-purpose computations.

## Chapter 1. References.

- [Blom83] R. Blomseth: "A Big RISC", Master's report, EECS, U. C. Berkeley 94720, June 1983
- [Fitz81] D. Fitzpatrick, J. Foderaro, M. Katevenis, H. Landman, D. Patterson, J. Peek, Z. Peshkess, C. Séquin, R. Sherburne, K. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer", VLSI Systems and Computations, Carnegie-Mellon Univ. Conference, Computer Science Press, pp. 327-338, October 1981. Also in: "A RISCy Approach to VLSI", VLSI Design, vol. II, no. 4, pp. 14-20, 4th qu. 1981; and in: Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.
- [FoVP82] J. Foderaro, K. VanDyke, D. Patterson: "Running RISCs", VLSI Design, vol. III, no. 5, pp. 27-32, Sep/Oct. 1982.
- [Henn82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill: "Hardware/Software Tradeoffs for Increased Performance", Proceedings, Symp. on Architectural Support for Programming Languages and Operating Systems, March 82, ACM SIGARCH-CAN-10.2 SIGPLAN-17.4, pp. 2-11.
- [Henn83] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor", Proceedings, 3rd Caltech Conference on VLSI, Pasadena, CA, March 83, Ed. R.Bryant, Comp. Sci. Press, pp. 33-54.

- [Kate80] M. Katevenis: "A Proposal for the LSI Implementation of the RISC I CPU (using a 3-phase clock)", Internal U.C.Berkeley Working Paper, 23 pages, September 1980.
- [Liou83] D. Lioupis: "The RISC II Computer", Internal U.C.Berkeley Working Paper, 25 pages, June 1983.
- [Orga82] E. Organick: "A Programmer's View of the Intel 432 System", McGraw-Hill, Hightstown, N.J., 1982.
- [Patt83] D. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, K. VanDyke: "Architecture of a VLSI Instruction Cache", Proc. of the 10th Symposium on Computer Architecture, ACM SIGARCH CAN 11.3, pp. 108-116, June 1983.
- [Radi82] G. Radin: "The 801 Minicomputer", Proceedings, Symp. on Architectural Support for Programming Languages and Operating Systems, March 82, ACM SIGARCH-CAN-10.2 SIGPLAN-17.4, pp. 39,47.
- [Stre78] W. Strecker: "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family", AFIPS Conf. Proc., Vol. 47, 1978 NCC, pp.967-980.
- [Tyne81] P. Tyner: "iAPX-432 General Data Processor Architecture Reference Manual", Order #171860-001, Intel, Santa Clara, Calif., 1981.
- [Utle78] B. Utley et.al.: "TBM System/38 Technical Developments", IBM GS80-0237, 1978.



## CHAPTER 2:

# THE NATURE OF GENERAL-PURPOSE COMPUTATIONS.

In the design of a computer system, two issues must be studied carefully:

(1) *FUNCTION*: What is the purpose of the computer system? What is the nature of the computations it will perform? What are the necessary features that will enable it to perform those computations with high efficiency?

(2) *COST*: Can the desirable architectural features be implemented at a reasonable cost and with a reasonable performance, in a particular technology? What are the trade-offs imposed by the constraints of a given implementation technology?

This chapter focuses on the first question of what it is that computer systems usually do, leaving the bulk of the discussion on implementation issues for the next chapters. We are interested in "general-purpose computer systems". Although it is difficult to define this term, we use it to refer to systems not biased towards the execution of a particular algorithm, and, specifically, systems that execute a mix of word processing, data base applications, mail and communications, compilations, CAD, control, and numerical applications. The chapter will assemble a picture of the nature of such "general-purpose"

computations, by collecting program measurements from the literature, and by studying the critical loops of some representative programs. The resulting picture will be used in the next chapters.

## 2.1 Goal and Methods of Program Measurements.

The main vehicle for a qualitative and quantitative understanding of the nature of computations is the measurement of the important properties on some real programs. It is very difficult for such a study to be made *abstractly* -- not in connection with a particular model of computers and computations, because real programs and programming languages are written and defined with a particular model in mind, and because the properties to be measured depend on this model.

Throughout this dissertation, a *von Neumann model* of computers and computations is assumed. Programs written in corresponding languages are considered in this chapter. C and FORTRAN program fragments are studied, and measurements from the literature are reported, which were collected by looking at programs written in FORTRAN, XPL, PL/I, Algol, Pascal, C, BLISS, Basic, and SAL. This section identifies the main properties of computations which are important in the design of von Neumann architectures, and lists tools and methods for their measurement.

### 2.1.1 Architecturally Important Properties of Computations.

In the von Neumann model, computations are performed by sequentially executing operations on operands which are kept in a storage device. The sequence of operations is dynamically controlled by operand values. Thus, the properties of computations that will interest us are:

- **Operands used.** Their type, size, structure, and the nature of their usage determines the storage organization for keeping them and the addressing modes for accessing them. In particular:
  - Constant or variable operands.
  - Types of operands: integers, floating-point, characters, pointers.
  - Structure of operands: scalars, arrays, strings, structures of records.
  - Declaration of operands: globals, procedure arguments, procedure locals.
  - Number of operands, sizes, and frequency of accesses for the above categories.
  - Amount and nature of locality-of-reference, possibly determined individually for each one of the above categories, for example, for scalars, arrays, (dynamic) structures, globals, and procedure activation records.
- **Operations performed.** These will determine the required operational units, and their connection to the storage units. The relative frequency of operations such as the ones listed below is important, and the variation of those frequencies with the operands' categories is also of interest.
  - Test, compare, add, subtract, multiply, divide, and so on.
  - Operation type, such as integer, floating-point, or string.
  - Higher level operations, such as I/O, buffer, list, and so forth.
- **Execution sequencing.** This will determine the control and pipeline organization:
  - Control transfers: conditional/unconditional jumps, calls, returns. What is their frequency, distance, conditions, predictability, and earliness of condition resolution.
  - Amount and nature of extractable parallelism. This is a very general and important question; for von Neumann architectures, we are interested in low-level parallelism.

While quantitative measurements are essential, the large number of properties to be measured -- especially if correlation among them is also studied -- makes a qualitative understanding of the global picture equally important.

Methods for both kinds of analysis are presented below.

### 2.1.2 Static and Dynamic Measurements.

Program measurements are usually collected by running the program under study through a suitable filter, or by executing it in a suitable environment. In both cases the result of this processing is a count of the numbers of times that some feature has appeared or that some particular property has held true in the text of the program or in its execution.

Measurements referring to the text of a program are called static. They give no useful information on performance, because they are not weighted relative to the number of times each statement was executed. They can show the size of storage required for the machine code and for the statically allocated objects, and they can show what the compiler has to deal with. Under crude assumptions, the static characteristics of programs can also give some indication on their dynamic behaviour.

Measurements referring to the execution of a program are called dynamic. Execution of the program requires previous compilation into object code for some machine, except if expensive interpretation is used. Thus, dynamic measurements usually refer to machine rather than source code, introducing another - often unwanted - parameter into the study. Machine code can be correlated back to source code, so that dynamic measurements at the source level can be inferred. However, this correlation is not always easy or precise.

Static and dynamic program measurements have frequently appeared in the literature, and have also been collected early in the RISC project (spring 1980). Section 2.2 reviews some of them.

### 2.1.3 Source-Code Profiling and Studying.

Because the list of important properties of computations is very long, and because several of them are difficult to quantify or to measure, the static and dynamic program measurements have some limitations. There is another method of looking at the nature of computations which is less quantitative but more qualitative, and which can complement these measurements or give a better idea of what specific other measurements should be taken. That method is to carefully study the source code of a program and, if possible, the underlying algorithms, concentrating on those portions of it which account for most of the execution time.

It has been observed, time and again, that programs spend most of their execution time in small portions of their code, the so-called "critical loops". This makes it feasible and worthwhile to study those portions in detail, to understand the nature and properties of the computation that is carried out. The critical loops can be identified by profiling the program during execution. Profiling is the dynamic measurement of how much of the execution "cost" is spent at each place in the program's code. The "cost" may be:

- time spent,
- number of source-code lines executed,
- number of memory accesses, and so forth.

In section 2.3 we will study some critical loops that have been identified by other researchers. Section 2.4 studies some more critical loops, which were identified by this author using two profiling systems. The first was the standard profiling facility of UNIX: compilation using the `-p` or `-pg` switch, execution, and then interpretation of the results by the `prof` or `gprof` program. This method arranges that the program-counter of an executing process be sampled at "random" intervals (on clock interrupts, every 1/60th of a second). The sampled

value is used to determine which procedure was executing at that time. If a program runs for a long time, the above samples can be used to construct estimates of how much time was spent in each of the program's procedures. There is no straightforward way to find out the time spent in executing any smaller program portions.

The second profiling system that was used, for programs written in C, belongs to Bell Laboratories (Murray Hill), and was used under special authorization [Wein]. It counts the number of times that each source-code line is executed (but gives no indication as to how long its execution takes). A special version of the C compiler is used, which inserts code at appropriate locations to increment appropriate counters. At the end of execution the counts are saved in a file. Another program is then invoked to correlate those counts with the original source code, and to generate an annotated program listing †.

## 2.2 Review of some Program Measurements from the Literature.

In this section interesting program measurements from the literature are reviewed. Measurements on all properties mentioned in section 2.1.1 are not present here, because some of them either have not received enough attention in the literature, or were difficult to measure. The measurements were selected from:

---

† the count is not always what one would expect for lines like: " } else { ". The listings in section 2.4 have been corrected by hand in those situations.



- [AlWo75]: Alexander and Wortman collected static and dynamic measurements from 19 programs (mostly compilers), written in XPL and executed on the IBM/360 architecture.
- [Elsh76] Elshoff presented static measurements of 120 commercial, production PL/I programs for business data processing.
- [HaKe80] [TaSe83] Halbert and Kessler, in their study of multiple overlapping windows early during the RISC project, collected dynamic measurements on the number of arguments and local scalars per procedure, and on the locality property of procedure-nesting-depth. They measured the C compiler, the Pascal interpreter, the troff typesetter, and 6 other smaller non-numeric programs (all written in C). Tamir and Séquin collected some more dynamic data on the locality of nesting depth, measuring the RISC C compiler, the towers-of-Hanoi program, and the Puzzle program (all written in C).
- [Lund77] Lunde used the concept of "register-lives" in his measurements. He analyzed half a dozen numeric-computation programs written in 5 different HLL's (2 FORTRAN versions, Basic, Algol, BLISS), plus some compilers, all running on a DECsystem10 architecture.
- [Shus78] Shustek studied the usage made of the PDP-11 addressing modes, by statically measuring 10,000 lines of code of an operating system.
- [PaSe82] Patterson and Séquin presented the most important measurements collected during the early stages of the RISC project, in spring 1980, in collaboration with E. Cohen and N. Soiffer. Measurements are dynamic, and were collected from compilers, typesetters, and programs for CAD, sorting, and file comparison. Four of those were written in C, and the other four in Pascal.
- [Tane78] Tanenbaum published static and dynamic measurements of HLL constructs, collected from more than 300 procedures used in operating-system programs and written in a language that supports structured programming (SAL).

### 2.2.1 Measurements on Operations.

The operations performed by programs are the most frequent object of measurements in the form of statement types (source level) or opcodes (machine level). The following table summarizes such measurements.

Property:	Measurem.:	Reference:
<i>Dynamically executed instructions:</i>		
moves between registers and memory	40 %	[Lund77,p.149]
branching instructions	30 %	(numeric & compilers)
fixed-point add/sub's	12 %	
load, load address	33 %	[AlWo75]
(more than normal, due to 360 architecture)		(mostly compil. in XPL on IBM/360)
store	10 %	
branch	14 %	
compare	6 %	
<i>Statically counted HLL statements:</i>		
assignments	42 %	[AlWo75]
if	13 %	(mostly compil. in XPL)
call	13 %	
<i>Dynamically executed HLL statements:</i>		
assignments	42 ± 12 %	[PaSe82]
if	36 ± 15 %	(non-numeric, in C and Pascal)
call/return	14 ± 4 %	
loops	4 ± 3 %	
<i>....weighted with the number of machine instructions executed for each:</i>		
loops	37 ± 5 %	[PaSe82]
call/return	32 ± 12 %	(non-numeric, in C and Pascal)
if	16 ± 7 %	
assign	13 ± 4 %	
<i>....weighted with the number of memory accesses necessary for each:</i>		
call/return	45 ± 16 %	[PaSe82]
loops	30 ± 4 %	(non-numeric, in C and Pascal)
assign	15 ± 5 %	
if	10 ± 4 %	
<i>More on procedure calls:</i>		
procedure calls as percentage of dynamically executed HLL statements	12 %	[Tane78] (O.S., structured prog.)
procedure call administration as percentage of execution time	25 %	[Lund77,p.151] (BLISS compiler)
<i>an amazing exception case:</i>		
procedures defined within 100 K statem.	83 (only!)	[Elsh76] (PL/I business prog., static)
perc. of calls relative to all statem.	2 % (!)	
<i>Other frequent high-level operations:</i>		
• vector operations (inner product, move, sum, search,...) [Lund77]		
• character-string ops (table-controlled substitute, delete, branch)		
• loop control (increment a reg., compare it to another reg., and branch)		

Property:	Measurem.:	Reference:
<i>Jump distance, measured dynamically:</i>		
< 128 bytes	55 %	[AlWo75]
< 16 Kbytes	93 %	
<i>Jump conditions, measured dynamically:</i>		
unconditional jumps as % of all jumps	55 %	[AlWo75]
... "the comparison of two non-zero values is about twice as common as comparison with zero".		[Lund77]
<i>Expressions, register lives:</i>		
one-term expressions in assignments†	68 %	[Tane78]
two-term expressions in assignments†	20 %	(dynamic)
operators per expression (average)	0.78	[AlWo75] (stat.)
relative to all register lives:		
lives w. no arithm. performed on them	50% (20-90%)	[Lund77]
lives w. max†† integer add/sub on them	25% (1-70%)	(dynamic,
lives w. max†† integer mult/div on them	5% (2-20%)	numeric &
lives used in floating-point operations	15% (0-40%)	compilers)
lives used for indexing	40% (20-70%)	[Lund77]

† on the right-hand-side of assignments.

†† "maximum-complexity" operation performed on the register,  
where int-add/sub < int-mult/div < floating-point-op.

These measurements are not very helpful in understanding the high-level nature of computations, but they do show:

- The importance of the procedure call mechanism, since so much time is spent in it.
- The importance of the sequencing control mechanism (compare and branch), since loops and if's are so frequent.
- The importance of simple arithmetic and of addressing, accessing, and moving operands around, since expressions are usually very short, and since half of the operands appearing in registers ("register lives" in [Lund77]) have no arithmetic performed on them.

## 2.2.2 Measurements on Operands.

Measurements on the operands in programs have not been so frequent in the literature, even though this subject is very important. Lunde [Lund77] measured on a DECsystem10 that each instruction on the average references 0.5 operands in memory and 1.4 in registers dynamically. These figures depend highly on the architecture and on the compiler, but they do illustrate,

nevertheless, the importance of fast operand accessing, since that occurs so frequently.

Property:	Measurem.:	Reference:
<i>Dynamic percentage of operands (HLL):</i>		
integer constants	$20 \pm 7 \%$	[PaSe82]
scalars	$55 \pm 11 \%$	(non-numeric,
array/structure	$25 \pm 14 \%$	in C and Pascal)
local-scalar references as percentage of all scalar references	$> 80 \%$	[PaSe82]
global-array/structure references as percentage of all arr/str. references	$> 90 \%$	[PaSe82]
<i>Use of PDP-11 addressing modes:</i>		
"The four most common modes are perhaps the four simplest":		[Shus78]
register	32 %	(static,
indexed (e.g. for fields of structures)	17 %	O.S.)
immediate (constants)	15 %	
PC-relative (direct addressing)	11 %	
all others	25 %	
<i>"The four least-used modes are precisely the 4 memory indirect ones (1%)".</i>		
"Half of the move instr. were moving something into a register"		[Shus78]
"Half of the compare/add/subtract instructions had one of their operands be an immediate"		

A property that had attracted very little attention in the past is the high locality of references to local scalar variables. The figures from [PaSe82] given above show that over half of the accesses to non-constant values are made to local scalars. On top of that, references to arrays/structures require a previous reference to their index or pointer, which is again a - usually local - scalar. Most of the time, the number of local scalars per procedure is small.

Tanenbaum [Tane78] found that 98 % of the dynamically called procedures had less than 6 arguments, and that 92 % of them had less than 6 local scalar variables. Similar numbers were found by Halbert and Kessler:

<i>Procedure Activation Records: [HaKe80]</i> Percentage of executed procedure calls with:		
	compiler, interpr. and typesetter	other smaller pro- grams (non-numeric)
> 3 arguments	0 to 7 %	0 to 5 %
> 5 arguments	0 to 3 %	0 %
> 8 words of arguments & locals	1 to 20 %	0 to 6 %
> 12 words of arguments & locals	1 to 6 %	0 to 3 %

Thus, the number of words per procedure activation is not large. The following measurements show that the number of procedure activations touched during a reasonable time span is not large either. This establishes the locality-of-reference property for local scalars.

<i>Locality of Procedure Nesting Depth: [HaKe80] [TaSe83]</i> Percentage of executed procedure calls which overflow from last span of nesting depths:		
(assuming that the span of nesting depths has constant size, and that its position moves by one on every over/under-flow; this corresponds to a RISC register file with as many windows as the span size, and with no window reserved for interrupts. See section 3.2).		
	2 compilers, interpr. typesetter, Hanoi	8+1 other smaller pro- grams (non-numeric)
span size = 4 (4 windows)	8 to 15 %	0 to 2.5 %
span size = 8 (8 windows)	1 to 3 %	0 to 0.2 %

## 2.3 Study of some Critical FORTRAN Loops (collected mostly by Knuth).

Knuth, in [Knut71], presents a study of where FORTRAN programs spend most of their time. The programs he measured varied from text-editing to scientific number-crunching programs. Dynamic measurements of the HLL

statements executed showed that:

- 67% were assignments,
- one third of those assignments were of the type  $A=B$ ,
- 11% were IF, 9% were GOTO, 3% were DO,
- 3% were CALL, and 3% were RETURN,
- More than 25% of the execution time was spent in I/O formatting.

However, what is most interesting for our study is that he gives the actual code fragments where 17 of those programs (chosen at random) spent most of their time. He used those fragments ("examples") to test the effectiveness of various techniques for optimization of compiled code. We will briefly study those same examples from our point of interest: understanding the nature of computations, and in particular answering the questions of section 2.1.1. The 17 examples have been classified in three categories of array-numeric, array-searching, and miscellaneous style examples. Their code (or a summary of it) is given below in a modernized-FORTRAN format. An eighteenth example of a critical loop, collected by the author of this dissertation, was added to the first category. It is the main loop of a procedure that inverts a positive-definite symmetric matrix. It was included in the study after two researchers in structural mechanics and in fluid dynamics independently told this author that they felt matrix inversion was the most time-consuming computation done by people in their area.

### 2.3.1 "Array-Numeric" Style Examples.

Example 3:     double A, B, D  
                   do 1 k=1,N  
 1    A = T[I-k, 1+k]; B = T[I-k, J+k]; D = D - A\*B

Example 7:     do 1 i=1,N  
                   A = X\*\*2 + Y\*\*2 - 2.\*X\*Y\*C[i]



```

1  B = SQRT(A) ; K = 100.*B + 1.5 ; D[i] = S[i] * T[K]
   Q = D[1] - D[N]
   do 2 i=2,M,2
2  Q = Q + 4.*D[i] + 2.*D[i+1]

```

Example 9:

```

do 2 k=1,M
do 2 j=1,M
initialize...
do 1 i=1,M
  N = j + j + (i-1)*M2 ; B = A[k,i]
1  X = X + B*Z[N] ; Y = Y + B*Z[N-1]
2  more computations...

```

Example 11: a Fast Fourier Transform. It computes sums and products of floating-point elements of two linear arrays. One array is accessed sequentially, and the other one with a step of N.

Example 12: a very long inner loop, with counter arithmetic, array accesses (many 3-dimensional arrays, some 2- and 1- dimensional), and floating-point multiplications and additions. There is one expression with 32 operators! In spite of its heavy computation character, this program has no more floating-point operations than it has simple counter and index operations.

Example 15:

```

do 1 j=i,N
  H[i,j] = H[i,j] + S[i]*S[j]/D1 - S[k+i]*S[k+j]/D2
1  H[j,i] = H[i,j]

```

Example 17:

```

do 1 i=1,N
1  A = A + B[i] + C[k+i]

```

Example - Matrix Inversion:

Figure 2.3.1 shows the aforementioned critical loop of positive-definite symmetric matrix inversion, in an abstract flow-chart form.

All these critical loops are of the same style: They perform floating-point operations on elements of arrays. Two almost independent "processes" exist. First, array elements are accessed in a *regular* fashion, i.e. in an arithmetic progression of memory addresses; the loop control is related to the array indexes, and does not depend on the array data. The second "process" is that of doing the actual numerical data computations.

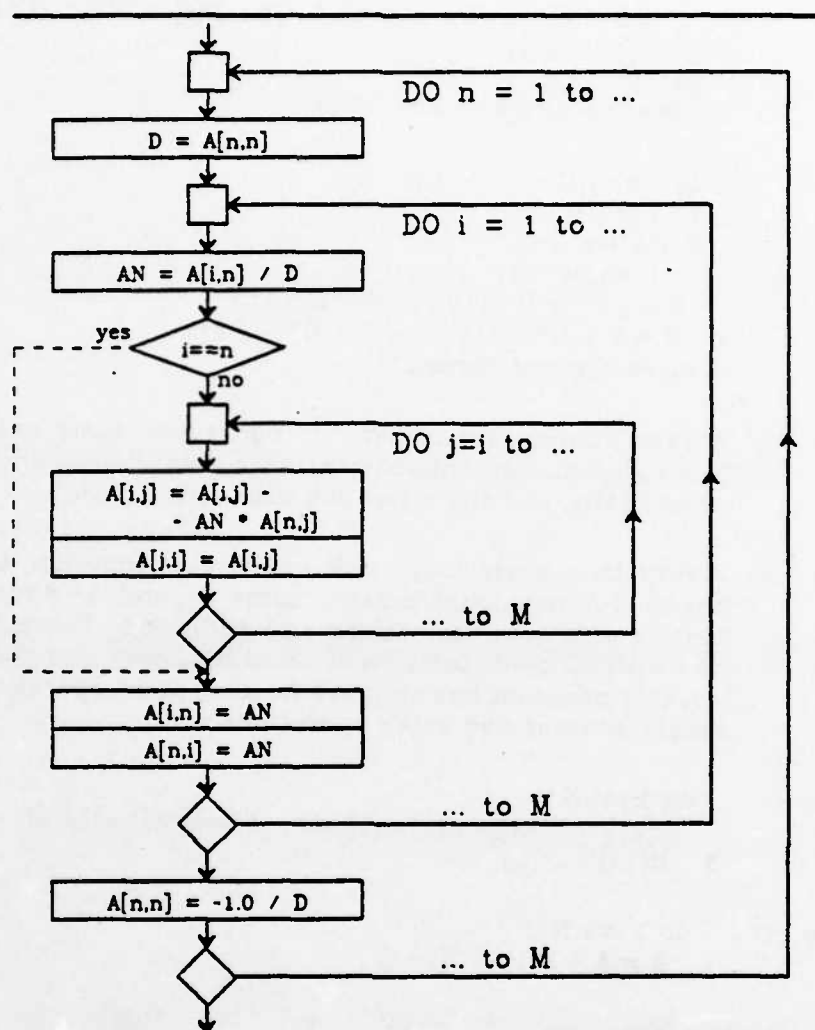


Figure 2.3.1: Main Loop of a Procedure to Invert a Positive-Definite Symmetric Matrix.

### 2.3.2 "Array-Searching" Style Examples.

Example 1: a search for the maximum of the absolute values:  
 do 2 j=1,N  
   t = ABS( A[i,j] ) ; if (t>s) then s=t ;  
 2 continue

Example 2: a search for a match:  
 do 1 j=38,53  
   if (K[i]=L[j]) then goto 2  
 1 continue

Example 10:   do 1 i=L,M  
               1 if ( X[i-1,j] < Q and X[i,j] ≥ Q ) then rare

Example 13: a binary search:  
               1 j = (i+k)/2  
               if (j==i) then goto 2  
               if ( X[j] == XKEY ) then goto 3  
               if ( X[j] < XKEY ) then i=j else k=j  
               goto 1

These examples are non-numeric. Most of them access the array(s) in a regular manner, like the examples in 2.3.1. However, the control of their sequencing is *dynamic* in nature: it depends on the actual data being visited, rather than on regularly incremented counters.

### 2.3.3 "Miscellaneous" Style Examples.

Example 4: first a poor quality random-number generator is defined:

```
subroutine RAND(R)
  j = i * 65539
  if (j<0) then j = j + 2147483647 + 1
  R = j; R = R * 0.4656613e-9
  i = j; k = k+1; return
```

then it is called:

```
do 1 k=M,20
  call RAND(R)
1 if ( R > 0.81 ) then N[k] = 1
```

Knuth comments: "...the most interesting thing here, however, is the effect of subroutine linkage, since the long prologue and epilogue significantly increase the time of the inner loop".

Example 5: this is a long inner loop that does lots of floating-point computations. It contains some simple arithmetic and compare & branch operations on integer counters, sequential addressing of two linear arrays, and several floating-point exponentiations, multiplications, and additions. The loop is badly written, with many large common subexpressions. There is lots of low-level parallelism present, mainly among the floating-point computations, but also between them and the integer ones.

Example 6: a subroutine S is defined:

```
subroutine S(A,B,X)
  dimension A[2], B[2]
  X=0; Y = (B[2]-A[2])*12 + B[1] - A[1]
  if (Y<0) then goto 1
```

```

      X=Y
1   return
then W is defined, which is called multiple times, and which calls S:
      subroutine W(A,B,C,D,X)
      dimension A[2], B[2], C[2], D[2], U[2], V[2]
      X=0 ; call S(A,D,X) ; if (X==0) then goto 3
      call S(C,B,X) ; if (X==0) then goto 3
      rarely executed code
3   return

```

Example 8:      subroutine COMPUTE ; common ....  
                  complex Y[10], Z[10]  
                  R=real(Y[n]) ; P=sin(R) ; Q=cos(R)  
                  S = C \* 6.0 \* (P/3.0 - Q\*Q\*P)  
                  T = 1.414214 \* P \* P \* Q \* C \* 6.0  
                  U=T/2.  
                  V = -2.0 \* C \* 6.0 \* (P/3.0 - Q\*Q\*P/2.0)  
                  Z[1] = (0.0,-1.0) \* ( S\*Y[1] + T\*Y[2] )  
                  Z[2] = (0.0,-1.0) \* ( U\*Y[1] + V\*Y[2] )  
                  return

Example 14:     do 1 i=1,N  
                  1   C = C/D\*R ; D = D-1 ; R = R+1

Example 16:     real function F(X)  
                  Y = X \* 0.7071068  
                  if ( Y < 0.0 ) then goto 1  
                  *rarely executed code*  
                  1   F = 1.0 - 0.5 \* (1.0 + ERF(-Y)) ; return

These examples help us remember that real programs are not always as simple and straightforward as those seen in sections 2.3.1 and 2.3.2. Relative to those simpler ones, these "miscellaneous" programs are characterized by more numeric computations, the same number or fewer array accesses, less index/counter arithmetic, less or unusual-style comparisons and branches, and — in some cases — more procedure calls.

### 2.3.4 The Nature of Numeric Computations.

The above examples give a picture of typical numeric computations, which can be summarized as follows:

1. The absolutely predominant data structure is the **array**. Most of the arrays are 1- or 2- dimensional. (Of course, the predominance of arrays over other data-structures can not be deduced by studying FORTRAN programs, since arrays are the only data-structure allowed in that language. However, it is known that the vast majority of numerical computations is performed to solve engineering or other similar problems, where the array arises as the natural data-structure.)
2. In the vast majority of the cases, the array elements are **accessed in regular sequence(s)**. There are a few "working locations" in the array(s), and their addresses change as arithmetic progressions. The step is quite often equal to one element size, or, at other times, it is the column size or some other constant.
3. A few integer scalar variables are used as **loop-counters and array-indexes**. The arithmetic performed on them is simple and corresponds to the above "regular sequence" of array accesses: increment by a constant, compare & branch. **Address computations** for multi-dimensional arrays require integer multiplication. Most of the times, it is feasible and advantageous for the optimizing compiler (or the very sophisticated programmer) to replace those integer counters/indexes by actual memory pointers; the address computations are avoided in this way (see [AhU177], p.466: Induction Variable Elimination).
4. The numeric computations are usually **floating-point operations** (multiplications and additions/subtractions being the most frequent). Several such operations are performed, but usually not many more in number than the integer operations on counters.
5. **Low-level parallelism** is present in many cases, and has two forms: (1) among various floating-point operations, usually when long expressions are computed,

and when a series of assignment statements is executed with no control-transfers in between; and (2) between counter/address calculations and floating-point operations, especially when program sequencing (if's, loop's) depend on the former only. This quite common "static nature" of program sequencing is an important characteristic of programs which perform a certain computation on all elements of a vector or of an array.

6. The last property also gives to these programs significant amounts of **higher-level parallelism**. Subsequent loop iterations are independent and could proceed in parallel. Some times, they are completely independent (Example 15 of section 2.3.1), so that a highly pipelined von Neumann processor could take advantage of them. Other times, they are less independent (Example 17 in section 2.3.1 would require a tree-organized addition); von Neumann architectures and languages typically cannot exploit that parallelism.

## 2.4                      A Study of four C Programs for Text Processing and CAD of IC's.

In this section we study the critical loops of four non-numeric programs, written in C and taken out of the Berkeley UNIX† and CAD environment:

*fgrep*      the UNIX program which searches a file for occurrences of fixed strings,

*sed*        the UNIX stream (batch) text editor,

---

†UNIX is a trademark of Bell Laboratories.



*sort* the UNIX program to sort the lines in a file, and

*mextra* a circuit extractor [FitzMe] which, given a description of the IC's geometry, generates a list of the transistors and their interconnections present in an integrated circuit. It works by first reading-in the description of the geometry and building a corresponding dynamic data structure, and then "scanning" the IC following horizontal scan-lines of gradually increasing y-coordinate. It may be considered an example of a program that *manipulates a non-trivial dynamic data structure*.

As an argument in support of the representativeness of the above sample of programs, let us look at a typical compiler. Kessler's Pascal compiler spends most of its time [Kess82] scanning the input (i.e. reading and recognizing characters), generating assembly code (i.e. character I/O), and walking through tree structures and interrogating them. These functions are similar to what *fgrep*, *sed*, and *mextra* do.

The tools described in section 2.1.3 were used for locating the critical loops. Below, wherever code is shown, the number on the left of each line is the count of how many times the line was executed during the test run.

### 2.4.1 FGREP: a String Search Program.

In the test run, *fgrep* was used to search for occurrences of the string "kateveni" in a file of size  $\approx$  230 KBytes (there were a few hundred such occurrences). The run took about 8 seconds CPU time, allocated as follows:

- $\approx$  87% in the procedure *execute()*,
- $\approx$  11% in *\_read* (i.e. in the operating system),
- $\approx$  2% in everything else.

The procedure *execute()* follows:

---

*fgrep: execute()* [87%]:

```
| # define ccomp(a,b) (yflag ? lca(a)==lca(b) : a==b)
| # define lca(x) (isupper(x) ? tolower(x) : x)
```

```

struct words {
    char inp, out;
    struct words *nst, *link, *fail;
} w[MAXSIZ];
int yflag;

....

1 execute(file) char *file;
{ register struct words *c;
  register int ccount;
  register char ch, *p;
  char buf[2*BUFSIZ];
  int f, failed; char *nlp;

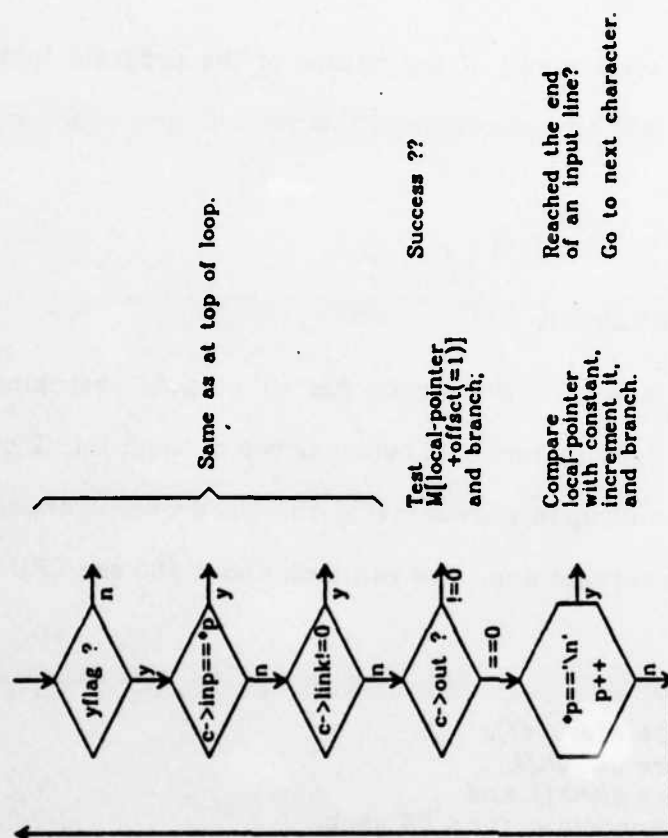
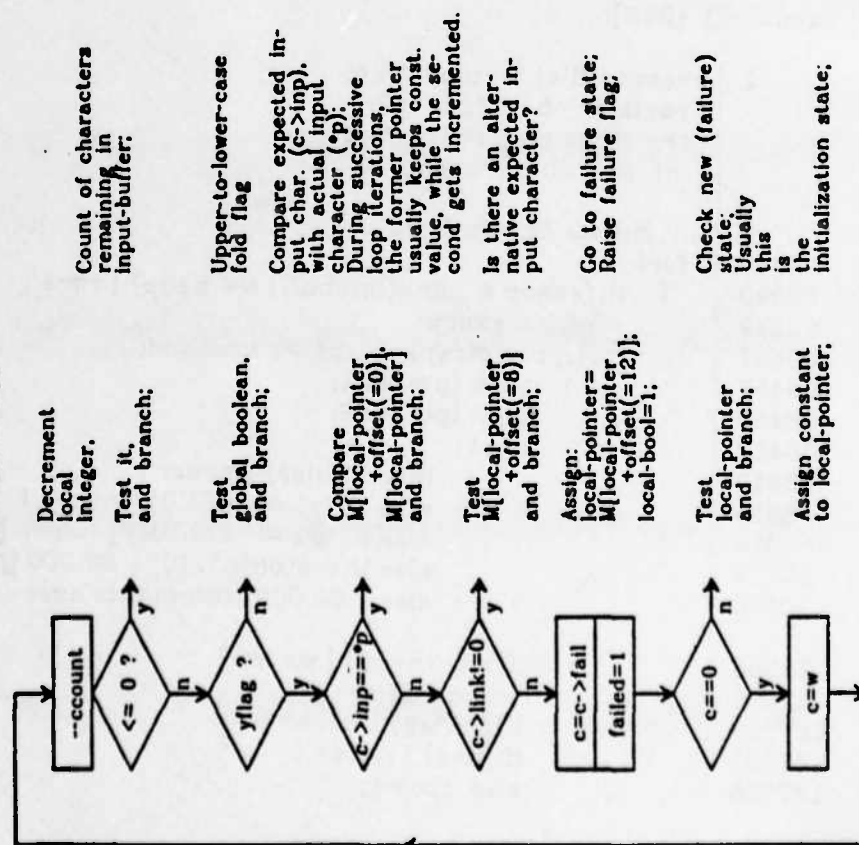
      .... Initial Set-Up Work ....
229253 for (;;)
229253 { if (--ccount <= 0)
228 { read-in a new 1Kbyte block or exit loop }
      nstate:
229252 if (ccomp(c->inp, *p)) /* in-line expansion */
923 { c = c->nst; }
228329 else if (c->link != 0)
0 { c = c->link; goto nstate; }
      else
228329 { c = c->fail;
228329 failed = 1;
228329 if (c==0)
228329 { c = w;
      istate:
228329 if (ccomp(c->inp, *p)) /* in-line exp. */
0 { c = c->nst; }
228329 else if (c->link != 0)
0 { c=c->link; goto istate; }
      }
0 else goto nstate;
      }
229252 if (c->out)
48 { Code for Success }
229204 if (*p++ == '\n')
4237 { Code for End-of-Line }
      }
1 .... Final Wrap-Up Work ....
}

```

Figure 2.4.1 contains a flow-chart of the critical loop of this run of *fgrep*. The vast majority of the operations performed are simply:

- accesses to scalars (mostly locals) and indirections through them to access fields of structures to which they are pointing, and
- comparisons (mostly to zero) & subsequent branches. The high frequency of

The local pointer *c* is pointing to the record identifying the current state of the FSM pattern-matching automaton. The local pointer *p* is pointing to the current input character in a 2Kbyte local buffer array; reading is done by 1Kbyte blocks.



Totals:

- 2 accesses to global scalars
- 16 accesses to local scalars (read's, write's, AND indirections through them are counted)
- 9 accesses through pointer + (short) offset
- 2 increment/decrement
- 3 compare-and-branch'es
- 7 test-and-branch'es

Figure 2.4.1: Critical Loop of fgrep.

compare-&-branches is in part a result of the nature of the program (pattern matching), but is also a general characteristic of the non-numeric programs, as the next examples will show.

## 2.4.2 SED: a Batch Text Editor.

In our test run, *sed* copies a 2.2 Mbyte file to output, searching for occurrences of three short fixed patterns. It replaces two of them with 2 others (one shorter, one longer), and upon encountering the third one, it appends a specified new line after the current one. The run took about 160 sec CPU time, allocated as follows:

- $\approx 23\%$  in the procedure *execute()*,
- $\approx 23\%$  in the procedure *match()*,
- $\approx 16\%$  in the procedure *gline()*, and
- all other procedures accounted for  $< 8\%$  each.

---

### *sed: execute()* [23%]:

```

1 | execute(file)  char *file;
   | { register char *p1, *p2;
   |   register union reptr *ipc;
   |   int c;  char *execp;
   |
   | 1 .... Initial Set-Up Work...
   | 52820 for(;;)
   | 52820 { if((execp = gline(linebuf)) == badp) { rare }
   | 52819   spend = execp;
   | 158457   for(ipc = ptrspace; ipc->command; )
   | 158457     { p1 = ipc->ad1;
   | 158457       p2 = ipc->ad2;
   | 158457       if(p1)
   | 52819         { if(ipc->inar) { never }
   | 52819           else if(*p1 == CEND) { never }
   | 52819           else if(*p1 == CLNUM) { never }
   | 52819           else if(match(p1, 0)) { 22,000 if's executed }
   | 30899           else { 62,000 statements executed; continue; }
   |         }
   | 127558   if(ipc->negfl) { never }
   | 127558   command(ipc);
   | 127558   if(delflag) { never }
   | 127558   if(jflag) { never }
   | 127558   else ipc++;

```

```

52819 |     }
2143025 |     if(!nflag && !delflag)
        |     { for(p1 = linebuf; p1 < spend; p1++)
        |         /* "spend" is a global pointer */
2143025 |         putc(*p1, stdout);
/* Note: in-line expanded to: ( -_job[1]._cnt>=0 ? *(_job[1]._ptr)++ = *p1 : {rare} ) */
/* _job[1]._cnt, and _job[1]._ptr are global scalars (the compiler knows their address */
52819 |         putc('\n', stdout);
        |     }
52819 |     if(aptr > abuf) { 22,000 calls: arout(); }
52819 |     delflag = 0;
        | }
    | }

```

---

Here, we have:

- 0.28 M procedure calls,
- 2.35 M compare-&-branch,
- 3.10 M test-&-branch,
- 4.40 M incrementations, and
- 0.50 M assignments with no operation (move-type).

The vast majority of operands are accessed indirectly, through local pointers with a zero or small offset. Other accesses are to local and global scalars. Certainly, a lot of this procedure's time is spent in the tight for loop that copies characters to standard output.

---

sed: match() [23%]:

```

161106 | match(expbuf, gf)  char *expbuf;
        | { register char    *p1, *p2, c;
161106 |
161106 |     if(gf) { Execute ≈ 150,000 statements }
158457 |     else { p1 = linebuf; locs = 0; }
161106 |     p2 = expbuf;
161106 |     if(*p2++) { never }
        |     /* fast check for first character: */
161106 |     if(*p2 == CCHR)
161106 |     { c = p2[1];
5242476 |         do { if(*p1 != c) continue;
289823 |             if(advance(p1, p2)) { infrequent }
5189445 |             } while(*p1++);
108075 |         return(0);
        |     }
        |     ... Various others, never executed...
    | }

```

---

sed: gline() [16%]:

```

52820 | char *gline(addr) char *addr;
      | { register char *p1, *p2; register c;
      |   ...Initial Set-Up Work (100,000 statements total)...
2174691 |   for (;;)
2174691 |     { if (p2 >= ebp) { rare ;
2174690 |       if ((c = *p2++) == '\n') { infrequent }
2121871 |       if(c) if(p1 < lbend)
2121871 |         *p1++ = c;
      |     }
      |   ...Final Wrap-Up Work (200,000 statements total)...
      | }

```

---

These two procedures spends most of their time scanning characters. *Match()* scans characters searching for some particular one. *Gline()* scans characters copying and checking them.

### 2.4.3 SORT: an Extreme, but Real Case.

The particular sorting program that was studied, namely the one installed on our UNIX machines, spent one third of the test run time in its calls to a trivial procedure *blank()* used to scan over blanks. Obviously, it is preferable that *blank()* were defined as a macro, so that it be expanded in-line. The test run consisted of sorting a 2.2 Mbyte file, relative to the second-in-line field and with elimination of duplicates. It took half-an-hour of CPU time.

sort: blank() [31%]:

```

28087970 | blank(c)
28087970 | { if(c==' ' || c=='\t')
6279488 |   return(1);
19808482 |   else return(0);
      | }

```

---

In general, text-processing programs spend a lot of their time in inner loops where they sequentially "walk" through the characters in buffers, copying, comparing, or testing various things.



It is important to notice that programs dealing with text waste a lot of memory bandwidth in the usual architectures, where a full memory word is accessed each time a byte transaction takes place.

Exploitation of parallelism is difficult in these programs, because of the high frequency of conditional branches. The amount of work done between two consecutive branches is usually quite small, with limited parallelism. Parallelism is often available between operations in two different blocks  $B_1$  and  $B_2$  separated by a conditional branch, where the branch *usually* follows the path that makes  $B_2$  execute after  $B_1$ . Programs are usually written in such a way that execution of  $B_2$  cannot start before it is certain that it should start. The programmer could rearrange the code and introduce temporary variables to hold tentative results, but doing so would lead to complicated and hard to maintain programs.

#### 2.4.4 MEXTRA: a Circuit Extraction Program.

*Mextra's* test run consisted of extracting the circuitry in the control section of the RISC II chip. It took 330 sec CPU time, allocated as follows:

- $\approx$  14% in the procedure *ScanSubSwath()*,
- $\approx$  11% in the procedure *Propagate()*,
- $\approx$  10% in the procedure *alloc()*,
- $\approx$  8% in the procedure *EndTrap()*,
- $\approx$  5% in the procedure *Free()*, and
- the remaining procedures took  $< 4\%$  of the total time each.

---

##### **mextra: ScanSubSwath() [14%]:**

```

771 | ScanSubSwath(bin)  int bin;
    | {  int i, newCount, n;
    |   register edge *new,*old,*last, *oldList,*newList;
    |
    |   ...Initial Set-Up Work (30,000 statements)...
353237 |   while(new != NIL && old != NIL)      /* NIL is 0 */
352488 |       {  if(new->bb.l < old->bb.l) { infrequent }
    |       else

```

```

302554 |         { if(n < old->bb.t)
254342 |             { if(last == NIL) { rare }
253628 |               else { last->next = old; last = old; }
254342 |               old = oldList;
254342 |               if(old != NIL) oldList = old->next;
           |           }
48212 |         else { infrequent }
           |       }
304254 |         if(depth[last->layer] == 0)
140442 |             StartTrap(last->layer,last);
304254 |         if((depth[last->layer] += last->dir) == 0)
139794 |             EndTrap(last->layer,last);
304254 |         nextEnd = (nextEnd < last->bb.t ? nextEnd : last->bb.t);
           |     }
           | ...Final Wrap-Up Work (250,000 statements)...
           | }

```

This procedure performs extensive list operations, using local pointers. The total operations performed in its critical loop are:

- 0.6 M procedure calls;
- 0.3 M additions (not counting address computations).
- 1.8 M test-&-branches;
- 1.0 M compare-&-branches;
- $\approx 0.6$  M accesses to a global scalar (nextEnd);
- 6.5 M accesses to locals (96% of them to pointers)  
(these include accesses for indirecting through them);
- 3.1 M accesses to fields of structures via a local pointer, and
- 1.0 M (random) accesses to a small array *depth*[10].

The basic pattern of memory accesses is the list traversal, which places a corresponding limit on locality-of-reference. However, during each loop iteration there are 11 accesses to fields of the structures pointed to by "old->" and by "last->". Accesses to various fields of the same structure are obviously accesses to neighboring memory locations, since the structure nodes here have a size of 8 words. Moreover, there are repeated accesses to the same field of the same structure, for example  $\approx 4$  accesses per iteration to "last->layer".

The available parallelism, is again limited by the high frequency of conditional branches. Some parallelism can be seen between accessing a memory location and computing the effective address for a subsequent memory access. For example:

```

if ( new->bb.l < old->bb.l )
if ( (depth[last->layer] += last->dir) == 0)

```

---

**mextra: Propagate() [11%]:**

```

773 | Propagate(y,yNext)  int y, yNext;
    | { int layer, height, tempx,tempy;
    |   register segment *above, *below, *next, *poly, *diff;
    |
    |   ...Initial Set-Up Work (8,000 statements)...
    |   for( above=Above[layer]; above!=NIL; above=above->next )
141443 |   {
    |     for( ; below!=NIL && below->right < above->left;
    |           below=below->next)
138000 |       if(below->area != 0) { rare }
    |       for( next=below; next!=NIL && next->left <= above->right;
    |             next=below->next)
138083 |         { below = next;
138083 |           if(above->node == 0)
    |             {
135024 |               above->node = below->node;
135024 |               above->area = below->area +
    |                 height * ( above->right - above->left - 1 ) / 100;
135024 |               above->perim = below->perim +
    |                 2 * (height + above->right - above->left -
    |                     MIN(above->right,below->right) +
    |                     MAX(above->left,below->left) ) / 10;
    |               /* Note: In-line expansions:
    |                 MIN(x,y) into: (x<y ? x : y); MAX(x,y) into: (x<y ? y : x) */
135024 |               below->perim = below->area = 0;
    |                 {
    |                   else { rare }
138083 |                   if(below->area != 0) { never }
    |                 }
141443 |               if(above->node == 0) { rare }
    |             }
    |           ...Final Wrap-Up Work (500,000 statements)...
    |         }
    |   }

```

---

Here again, extensive list operations are performed. The list-nodes have a size of 8 words, and are accessed via local pointers. During each loop iteration, 18 accesses are made to fields of a certain list-node, and 15 to fields of another. Each individual field is accessed an average of 3 times. This procedure has more numeric computations than the other procedures in this section, but these are still not the dominant factor.

---

mextra: alloc() [10%]:

```

283165 | alloc(n)
      | { register int tmp; register struct cell *ptr;
283165 |
283165 |     if(n<CELLSIZE-4) { rare }
283165 |     n = (n+WORDSIZE-1)/WORDSIZE; /* WORDSIZE is 2 in this example */
283165 |     if(TBLSIZE<=n) { rare }
283138 |     else if(FreeTbl[n]!=0)
258662 |     { ptr=FreeTbl[n]; FreeTbl[n]=ptr->next; --FreeCnt[n];
258662 |       if(ptr->status!=FREE || ptr->count!=n) { never }
258662 |       if(FreeCnt[n]!=0)
241417 |         { if(FreeTbl[n]->status!=FREE) { never }
241417 |           if(FreeTbl[n]->count!=n) { never }
      |         }
      |       else { rare }
      |     }
      |     else { infrequent }
283165 |     ptr->status = ALLOC; ptr->count = n; tmp = (int) ptr;
283165 |     if(n<TBLSIZE) AllocCnt[n]++;
283165 |     return(tmp+4);
      | }

```

---

This last procedure has no loop; it is entered many times, and does a little work each time. Besides accessing fields of structures via pointers, it also makes many references to the  $n^{\text{th}}$  elements of several arrays. These latter are *not* sequential array-element accesses. However, if the information were kept in a single array of structures, instead of in multiple simple arrays, then the above accesses would all be to neighboring memory locations. Slightly more parallelism can be found here, for example:

{ptr->status=ALLOC; ptr->count=n; tmp=(int)ptr; if(n<TBLSIZE) ...}.

Also, notice that the *if*'s that lead to *then-clauses* which never get executed are consistency checks, and they could all be done in parallel if the language allowed some way of expressing that.

The overall picture from this CAD program is one of many conditional branches and of many accesses to fields of structures using local pointers pointing to them. Although the application has some arithmetic that needs to be done, it does not play a dominant role. There are very few increment

operations, contrary to the previous programs studied in earlier sections, because this program deals with dynamic data structures. The locality-of-references to the elements of the data structures stems from the computation pattern of performing several accesses to various fields of a few structure instances, before interest shifts to some new such instances.

## 2.5 Summary of Findings.

In this chapter, we first reviewed static and dynamic program statistics collected by other researchers. Their results indicate that the simplest operations are also the ones that are executed most of the time.

Then, we looked at several FORTRAN programs, most of them doing numerical computations. We observed that they perform primarily floating-point arithmetic operations on operands which frequently are elements of arrays. The inner loops usually traverse the arrays in a "regular" fashion, using indexes that are incremented by a constant amount and compared to a limit. The use of pointers rather than indexes, by the programmer or by the optimizing compiler, would be advantageous.

Then, we studied some text-processing programs written in C, and saw that they spend a large fraction of their time running sequentially through character buffers. These are array elements, again, but here programmers usually access them indirectly through local pointers. The dominant operations are not arithmetic any more -- they are tests or comparisons for branching and mere copying.

Finally, we analyzed a program for CAD of IC's, which manipulates a non-trivial dynamic data structure. The fields of a few nodes (structures) are accessed several times indirectly through local pointers, before the program shifts its attention to some other nodes linked to the previous ones. Again, we found high frequencies of test/compare-&-branch and of copying.

In all cases, we saw that programs are organized in procedures and that procedure calls are frequent and costly in terms of execution time. Procedures usually have a few arguments and local variables, most of which are scalars, and are heavily used. The nesting depth fluctuates within narrow ranges for long periods of time.

We found low-level parallelism although usually in small amounts, mainly between address and data computations. The frequent occurrence of conditional-branch instructions greatly limits its exploitation.

General-purpose computations, as usually expressed in von Neumann languages, are carried out by walking through static or dynamic data structures in some - usually regular - path. Operand addressing, copying, and comparing for decision making, are factors of prime importance. Procedures are heavily used for hierarchical organizations. Numeric computations are frequent and expensive in some applications.

In the next chapters, possible architectural features for exploiting these program characteristics will be presented.



## Chapter 2.

## References.

- [AhUl77] A. Aho, J. Ullman: "Principles of Compiler Design", Addison-Wesley Publishing Co, 1977.
- [AlWo75] G. Alexander, D. Wortman: "Static and Dynamic characteristics of XPL programs", IEEE Computer magazine, Nov. 1975, pp.41-46.
- [Elsh76] J. Elshoff: "A Numerical Profile of Commercial PL/I Programs", Software - Practice and Experience, vol.6, 1976, pp.505-525.
- [FitzMe] D. Fitzpatrick: *Mextra*: a Manhattan circuit extraction program, U. C. Berkeley. See for example: "1983 VLSI Tools -- Selected works by the original artists", report No. UCB/CSD-83/115, Comp. Sci. Div., U. C. Berkeley, CA 94720, March 1983.
- [HaKe80] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames", CS292R-course final report, Univ. of California, Berkeley, June 1980.
- [Kess82] P. Kessler: private communication, U.C.Berkeley, August 1982.
- [Knut71] D. Knuth: "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, vol. 1, 1971, pp.105-133.
- [Lund77] A. Lunde: "Empirical Evaluation of some Features of Instruction Set Processor Architectures", Communications of the ACM, 20.3, March 77, pp.143-153.
- [PaSe82] D. Patterson, C. Séquin: "A VLSI RISC", IEEE Computer Magazine, vol.15, no.9, Sept. 1982, pp. 8-21.
- [Shus78] L. Shustek: "Analysis and Performance of Computer Instruction Sets", Doctoral Dissertation, Stanford University, 1977 or 78.
- [Tane78] A. Tanenbaum: "Implications of Structured Programming for Machine Architecture", Communications of the ACM, 21.3, March 78, pp.237-246.
- [TaSe83] Y. Tamir, C. Séquin: "Strategies for Managing the Register File in RISC", IEEE Transactions on Computers, vol. C-32, no. 11, November 1983.
- [Wein] P. Weinberger, *lcomp* compiler for profiling, Bell Laboratories, Murray Hill.

## CHAPTER 3:

# THE RISC I & II ARCHITECTURE AND PIPELINE.

In chapter 1 the complexity-speed trade-off was discussed, and the importance of effective utilization of hardware resources was stressed. In chapter 2 we observed the predominance of operand addressing and accessing, of comparisons, and of conditional branching in general-purpose computations.

In this chapter the architecture, the pipeline, and the basic timing of RISC I and II are presented. Architecture and micro-architecture discussions are intermixed because an understanding of the implementation is essential in making architectural decisions that lead to a high-performance processor. It is shown how the RISC I & II architecture efficiently supports integer general-purpose computations with a reduced instruction set, allowing for compact and fast implementation. Benchmark measurements of RISC's performance are reviewed.

The next two chapters deal with the design, layout, debugging, and testing of RISC II, while chapter 6 discusses possible hardware enhancements to RISC-style processors, for increased performance. A detailed and exact description of the RISC II architecture - for reference purposes - can be found in Appendix A.

### 3.1 The RISC I & II Instruction Set.

The architecture of the Berkeley RISC is register-oriented, because program measurements indicate that supporting fast operand accesses is of utmost importance. On the one hand, the compiler by default allocates some frequently used program variables into registers. On the other hand, operations onto operands in memory are decomposed into their orthogonal subtasks of first bringing the operands into registers, then performing the operation, and last moving the result to memory. This decomposition brings no loss in performance when proper pipelining is utilized, while it simplifies the instruction set and its implementation.

All RISC instructions have a fixed width of one word for simplicity and efficiency of the instruction fetch-and-sequence mechanism. The instruction format is simple, with fields at fixed locations, for simple and fast instruction decoding. The operations performed by the instructions all fit within the same general framework, allowing for a simple and fast data-path, for high utilization of the data-path resources, and for a simple homogeneous timing scheme.

RISC is a 32-bit architecture, since a 4 Giga-Byte virtual-address space is believed to be enough for the next several years. Bytes, half-words, and full-word integers (32 bits) are supported in memory; they are all converted into full-words when moved into registers. This offers simplicity, while maintaining full operational flexibility with integers and characters.

This section describes and discusses the RISC instruction set, but does not rigorously define it. Refer to Appendix A for an exact definition.

### 3.1.1 Register-Oriented Organization.

From one point of view, a computer does 70% operand accesses and 30% operations: for each operation one or two source operands are required, and the result is placed into another operand. When one also considers the high frequency of A-gets-B type assignments (§ 2.2.1 [Tane78], [AlWo75]; beginning of sect. 2.3; sect. 2.4), where there are operand references but no operation, one realizes how important it is for a computer system to have quick access to operands. The fastest storage device is a CPU register, not only because the register file is physically small and on the same chip as the CPU, but also because addressing is made with a much shorter address than for cache or memory. For these reasons, RISC tries to keep as many of its operands as possible in registers. It is not enough to keep only temporary unnamed results in the registers, because expressions are usually very short (see above references), and hence there are not too many such intermediate results. The latter is also the reason why an expression-evaluation stack and a stack architecture were not chosen for RISC. The Berkeley RISC architecture has many registers, organized in multiple overlapping windows, and its compiler by default allocates scalar arguments and local variables of procedures in them. Multi-window register files will be discussed further in sections 3.2, 6.1, and 6.2.

In RISC I and II, operations are performed by 3-operand register-to-register instructions:

$$R_d \leftarrow R_{s1} \text{ op } S2$$

Besides variables, immediate constants are also quite important in computations. In sect. 2.2.2 we saw that they account for 15 to 20 % of the operands used. Thus, in the above generic instruction, the second source operand  $S2$  can be either a register  $R_{s2}$  or an immediate constant  $imm$  (see sect. 3.1.4). One of the registers, namely  $R_0$ , always contains the hardwired constant zero. Writing

into it is allowed, but will *not* change its value.

The available operations *op* are:

- integer addition (without or with carry),
- integer subtraction (without or with carry),
- integer inverse subtraction ( $-R_{s,1} + S2$ ) (without or with carry),
- bitwise boolean *AND*, *OR*, *Exclusive-Or*,
- shift left-logical, right-logical, or right-arithmetic (all by an arbitrary amount).

All these instructions can optionally set the 4 existing condition codes (CC's). The add/sub instructions assume 32-bit signed 2's-complement operands. However, there are conditions for branching on, which will act as if the operation (comparison) were between unsigned 32-bit quantities. The *with-carry* versions of add/sub can be used for multi-word precision arithmetic. The shift instructions will shift  $R_{s,1}$  by the amount (0 through 31 bit-positions) specified in the 5 least-significant bits of  $S2$ . The logical shifts fill the emptied bit positions with zeros, whereas the arithmetic shift-right sign-extends the leading bit. Rotates and arithmetic shift-left are not included, because they do not exist in HLL's. Shifts by arbitrary amounts (more than 1 or 2 bit-positions) are *not* frequent in HLL's. Thus, their inclusion into our instruction set was contrary to the RISC philosophy; section 4.2 will show the negative consequences of this decision.

Several general and frequent operations, which do not appear explicitly in the above list, can readily be synthesized using the options available:

<i>Instruction:</i>	<i>Method of Synthesizing it:</i>
move	$R_d \leftarrow R_s + R_0$
increment, decrement	use add with immediate constant of 1, -1
complement	$R_0 \leftarrow R_s$
negate ( <i>NOT</i> )	$R_s \text{ XOR } "-1"$
clear	$R_d \leftarrow R_0 + R_0$
compare, test	use $R_0$ as $R_d$ , and set condition codes (CC's).

### 3.1.2 Memory Accessing, and Addressing Modes.

In RISC I & II all arithmetic, logical, and shift instructions operate on registers. Only the *load* and *store* instructions can access operands in memory and move them to/from registers. This simplifies the processor's data-path and control, the instruction format, and the handling of interrupts caused by demand-paging. Related performance issues are discussed in § 3.3.2 and 3.3.3.

Load and store instructions have a single addressing mode:

$$R_d \leftrightarrow M [ R_{s1} + S2 ]$$

The result of a RISC add instruction is used as effective address for a memory access. This single addressing mode, which matches well with the rest of RISC's instructions, is quite versatile and permits one to synthesize many other addressing modes:

<i>Mode:</i>	<i>HLL usage:</i>	<i>Synthesizing it in RISC:</i>
Absolute or direct	global scalar	$M [ R_B + imm ]$ (within $\pm 4$ Kbytes of base $R_B$ )
Register indirect	pointer deref. (*p)	$M [ R_p + R_0 ]$
Indexed	field of struc. (p->field)	$M [ R_p + field\_offs ]$
Indexed	linear byte array (a[i])	$M [ R_a + R_i ]$ (assume $R_a$ points to the base of a[])

Notice that the last mode can only be applied to byte arrays and not to arrays of half- or full-words, because no scaling by 2 or 4 is done on  $S2$ . The lack of such scaling also reduces the range of addresses accessible with the 13-bit  $S2$  immediate offset. The reason for this lack is that there is no circuit in RISC I & II for both shifting and adding in one instruction. The modification proposed in section 4.3 would amend this situation.

The RISC II implementation has one notable exception from the above uniform addressing scheme: the second source  $S2$  must be an immediate constant for store instructions; thus, the last addressing mode in the table cannot be



synthesized for store instructions. The reason has to do with implementation. The register file has two ports because all instructions read two source operands from it. A store instruction with a register- $S2$ , however, would need to read three registers:  $R_d$ ,  $R_{s1}$ , and  $S2$ . This could not be accommodated without major penalties for the data-path, neither could  $R_d$  be read conveniently at a later time. Since that addressing mode is not important enough to justify such penalties, the feature was left out, in accordance with the RISC concept.

Memory addresses in RISC are byte addresses. Half-word quantities are aligned on half-word boundaries, and full-word quantities on full-word boundaries. Half-words and bytes are always right-adjusted when they are in registers. The load and store instructions have different versions for full-word, half-word, and byte transfers. These versions perform the necessary change in alignment between memory and registers. The store instruction assumes that the memory system is capable of selectively writing into some of the 4 bytes of a word. Different versions of the load instruction exist for bringing signed or unsigned short quantities into registers with sign-extension or zero-filling.

The versatile addressing mode of the memory access instructions is also used for the control-transfer instructions (jump, call, return) in RISC I & II. However, because the Program Counter (PC) is not in the register file, a separate PC-relative addressing mode was added for control transfers:

$$\text{effective\_address} = PC + \text{imm}$$

Once that mode existed for jump/call/return instructions, it required a trivial hardware extension to use it for load and store instructions as well. This was done to allow the generation of relocatable code for separately compiled modules. Global data can be allocated next to the code, and referenced relative to the PC.

However, later this turned out to be a bad idea. One wants to keep code and data separate for allowing shared read-only code-segments, and for being able to have separate instruction and data caches (sect. 6.3, 6.4). PC-relative data accesses also preclude the remote-PC scheme (sect. 6.3). Finally, the use of a linkage editor does not pose any serious problems when the code is not relocatable.

### 3.1.3 Delayed Control Transfer.

The Berkeley RISC architecture was designed with pipelining in mind from the very beginning. In particular, overlapping of instruction fetch and execution was assumed. This pipeline is disrupted by control transfer instructions, such as conditional and unconditional jumps, procedure calls and returns. Figure 3.1.1 shows a control-transfer instruction  $I_1$  being fetched and then executed. During its execution, it computes the address of its potential target and evaluates the condition for conditional branching. Simultaneously, the subsequent instruction  $I_2$  is being fetched. When execution of  $I_1$  has completed, the fetching of its target  $I_3$  can start.

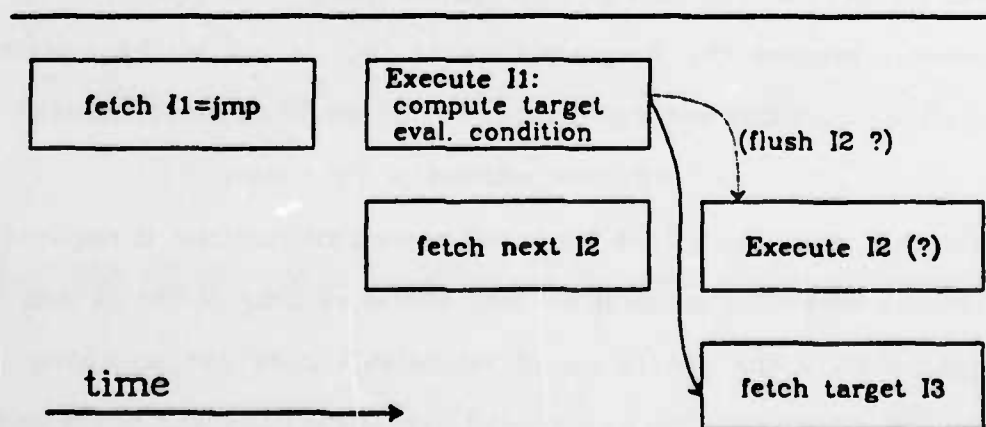


Figure 3.1.1: Delayed-Branch Scheme.

Instead of flushing  $I_2$  from the pipeline, and thus wasting one cycle, RISC employs the *delayed-branch* scheme. In that scheme, the transfer of control to  $I_3$  takes effect with a delay of one instruction.  $I_2$  is executed regardless of whether the control-transfer is successful or unsuccessful. Thus, an instruction immediately after a jump/call/return effectively belongs to the block preceding the transfer-instruction. The compiler puts a no-op at that place, while the optimizer tries to move a suitable task to that place. This can be done when the transfer-instruction does not depend on that task.

Measurements have shown [Camp80] that the optimizer is able to remove about 90% of the no-ops following unconditional transfers and 40% to 60% of those following conditional branches. The unconditional and conditional transfer-instructions each represent approximately 10% of all executed instructions (20% total). Thus, while a conventional pipeline would lose  $\approx 20\%$  of the cycles, optimized RISC code only loses about 6% of them. These rough calculations assume that most RISC instructions execute in one cycle (which is not far from true). The above figure agrees with a similar figure given by John Cocke of the IBM Watson Research Center during an informal discussion [Cock83]; the two-cycle branches executed by optimized 801 programs are about 6% of all executed instructions.

For the small fraction of the cycles that the optimizer cannot utilize, an actual no-op instruction has to be inserted at the place of  $I_2$ , consuming some code space. The current RISC I & II architectures have no special versions of the transfer-instructions that automatically suspend execution during the next cycle. This choice was made for simplicity. In retrospect, we could have reduced code size by 8% by adding the suspension capability with minimal penalty. The area penalty would be about 0.1% for a circuit that flushes  $I_2$  from the input of the opcode-decoder and replaces it with a no-op. There would be no

time penalty because that decoder is still small enough so that it does not affect the critical timing path (§ 4.3.3).

Control-transfer instructions use the same addressing modes as load's and store's. PC-relative is the preferred mode for jumps within a procedure, while register-indexed jumps can be used for table-driven case statements. Call instructions save the value of the PC into a register. The return instruction is register-indexed only; it uses the contents of the register where the corresponding call had saved the PC. In RISC II the return instruction is a conditional one, just like jumps. The usefulness of such instructions is very limited, but their implementation resulted quite naturally. Later, it turned out that conditional returns interfere with the critical path of interrupt assertion because an overflow trap should not occur on an unsuccessful return. This is another case where deviation from the RISC concept led to implementation penalties.

### 3.1.4 Fixed Instruction Format.

An important contribution to processor complexity comes from instruction decoding, and, in particular, from the task of extracting the various instruction fields. RISC I & II have a simple instruction format, with fixed field positions. This led to a very simple and fast decoding circuit (§ 4.3).

All RISC instructions are full-words of 32 bits. This greatly simplifies the instruction fetch and decoding task. Figure 3.1.2 shows the two instruction formats employed. Thirty-two registers are visible to the compiler at any one time, and thus a 5-bit field is necessary for specifying the sources and the destination. There is space for 128 opcodes, although only 39 of them are currently used. One bit (SCC) in every instruction can specify the optional setting of the condition codes according to the result ( $R_d$ ) of the instruction.

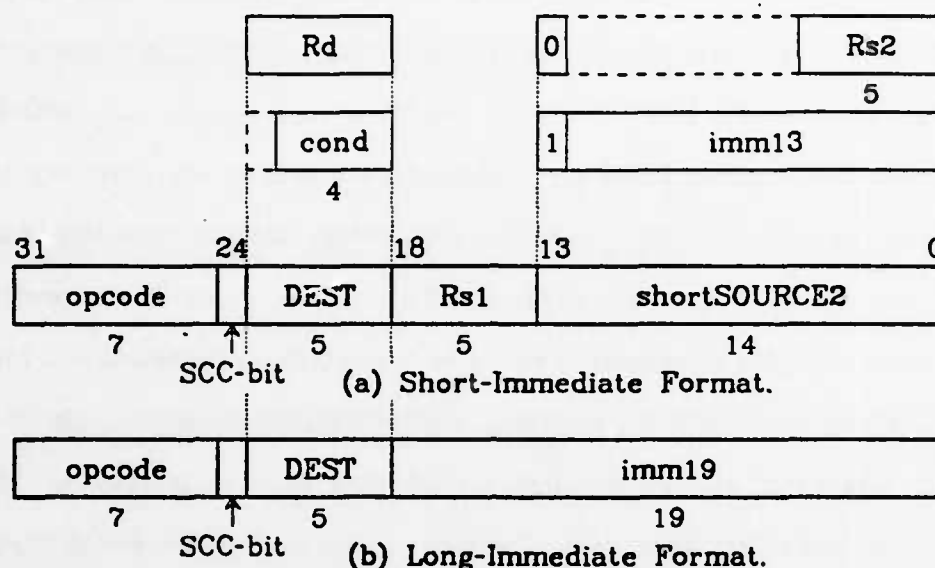


Figure 3.1.2: Instruction Formats.

The **DEST** field may specify one of two things in both instruction formats, according to the opcode. For conditional control-transfer instructions, its 4 least-significant bits specify the branch-condition (its MS bit is unused). For all other instructions, the **DEST** field specifies the  $R_d$  register number.

The short-immediate format is used for all register-to-register instructions and for register-indexed load, store, and control-transfer instructions. The shortSOURCE2 field consists of the 14 bits that are left over after the assignment of the other fields. Its leading bit specifies whether it should be interpreted as  $R_{s2}$  or as an immediate constant. In the former case, 8 of the 14 bits in the field are discarded (wasted). In the latter case, a 13-bit signed 2's-complement immediate constant is assumed.

The long-immediate format is used for all PC-relative instructions. Since the PC is the first source for them, these instructions need no  $R_{s1}$  and can have a wider immediate constant.

This format is also used for the *load-high* instruction, which takes a 19-bit immediate and places it into the 19 high-order bits of  $R_d$ , simultaneously zeroing the 13 low-order bits. *Load-high* can be used, in conjunction with the 13-bit immediate field of the following instruction, for loading any arbitrary 32-bit constant into a register. This method of introducing arbitrary constants into registers requires 64 bits of code space and 2 execution cycles. It was preferred over the more complex alternative relying on a single, longer instruction format that could hold the whole 32-bit constant. Since the memory bus is only 32-bits wide, two cycles would still be required for fetching such an instruction; thus there would be no performance gain. The size of that instruction would have to be 64 bits for proper alignment in memory, and no gains in code size would result either. Finally, a PC-relative load instruction can be used for the same purpose, but that means that parts of a code memory segment are read as data (see end of § 3.1.2).

### 3.1.5 Lack of String, Multiply, Floating-Point Support.

RISC I & II have no support for character-string operations, integer multiplications or divisions, or any kind of floating-point operations. There are various reasons for this.

Hardware support of some of these functions requires considerable silicon area, for instance a parallel multiplier, a floating-point unit, or support for more sophisticated string operations. If such a unit were included in the *central* data-path, the basic cycle time would be severely lengthened, due to increased size and capacitance. In § 4.2.4 we will see how even the moderately-sized shifter slows down the basic machine cycle. Another alternative is to place these units on the CPU chip, but outside the central data-path. This would make access to them slower than access to the integer adder, but it wouldn't



appreciably slow down the other operations.

A third alternative is to place special hardware *off* the CPU chip, for instance as a *co-processor*. Due to chip area limitations, the latter is the most attractive solution for today's technology. According to the views expressed in chapter 6, the integration of functional units such as an instruction cache on the CPU chip is very desirable and has higher priority than the integration of a large arithmetic unit. That means that the co-processor solution will remain attractive during the next few years as well. Co-processor architecture and interfacing is a large and important research area, which could not be undertaken as part of the Berkeley RISC project. For these reasons, RISC I & II have no parallel multiplier or floating-point hardware. Partial support for integer multiplication in the form of one step of Booth's algorithm is a feasible and attractive solution. The main reason why this was not included in RISC I & II is the lack of man-power for their design.

The situation for character string operations is different. These have not yet been standardized in the High-Level-Languages themselves. Most C programs perform them at the lowest character-by-character level (see for example the procedure *gline* in sect. 2.4.2). Pascal does not even have variable-size strings -- it merely has fixed size character arrays. Under these circumstances, it obviously makes no sense for the hardware to support something that the HLL itself does not support. This is, nevertheless, a very interesting area for future research and standardization. It is wasteful, in terms of memory bandwidth, to deal with strings at the character-level. One could decide to align all strings on word boundaries, and mark their end by null-byte padding in the last word (one or more null bytes to fill the word). Co-processors are well suited for supporting string operations as well, since strings are most likely to be kept in memory, and a co-processor hanging off the memory bus can process them as they go by

on the bus.

### 3.2 The RISC I & II Register File with Multiple Overlapping Fixed-Size Windows.

The importance of fast operand accesses and the desirability of keeping as many of them as possible in CPU registers were presented at the beginning of § 3.1. Registers are few in number, and instructions address them directly by their name. For both of these reasons they can only be used to hold scalar variables. One way of deciding which scalars to keep in registers is to rely on hints from the programmer, as are available in the language C. For global variables, this is probably the only way the compiler can know which ones to allocate in registers since programs usually have more global variables than the machine has registers.

But the situation is different for local variables. The measurements of Tanenbaum, and of Halbert and Kessler given in § 2.2.2 show that, for more than 95% of the dynamically called procedures, 12 words of storage are enough for all their arguments and locals. Thus, it is feasible for an architecture to have enough registers so that the compiler can allocate local scalars into registers by default. In case not all of them fit, the compiler will simply place the remaining variables in memory. The decision is not critical since the latter cases are so rare. The measurements from [PaSe82] reported in § 2.2.2 showed that out of 100 HLL operand references, about 20 were to constants, 55 to scalars, and 25 to arrays/structures. We can exclude constants from our count, since they are accessed as part of the instructions themselves. We can also safely assume that for every HLL array/structure reference there is also at least one access to a

scalar at the machine level: the array index or pointer to structure. Thus, a more representative ratio may be:  $55+25=80$  scalar accesses versus 25 non-scalar ones. Data from [PaSe82] and from our own measurements reported in § 2.4, indicate that about 80% of the scalar references are to local scalars. Thus, about 60% of all accesses are made to local scalars, and about 40% of them access all other kinds of objects. Since so few words are accessed with such a high frequency and with direct addressing, allocating them into registers is the obvious way of providing fast access to them.

The problem with keeping locals in registers is that they have to be saved on every procedure call and restored from memory on every return. This is the main source of the very high cost of procedure calls in terms of execution time (25 to 40 %, § 2.2.1: [PaSe82] [Lund77]). Argument passing is the second main source of cost. But, while procedure calls occur frequently, roughly once every 8 HLL statements (§ 2.2.1: [AlWo75] [PaSe82] [Tane78]), strong variations of their dynamic nesting depth are rarely observed. This locality of the nesting-depth means that, if sufficient register storage is provided for a few activation records, instead of only for one, then register saving and restoring can be reduced dramatically. This led Halbert and Kessler to propose a large register file with multiple overlapping windows for the RISC architecture [HaKe80]. Previous proposals on this subject had been made by R. Sites [Site79], and F. Baskett [Bask78], although their schemes differed from the one used in RISC I & II. Multiple register windows had appeared in processors before RISC, but they were usually intended for multiple processes rather than for procedures, or they had no overlap. More measurements and studies on multiple windows can also be found in [DiML82] and [TaSe83]. The latter paper studies the problem of optimally managing the RISC register file.

### 3.2.1 Overlapping, Fixed-Size Windows.

Figure 3.2.1 (a) shows the organization of a register file into fixed-size, overlapping windows. Not all CPU registers are simultaneously visible by the machine language programmer at any given time. The ones that are visible are called "the current window". The window-number inputs to the decoder select the current window. They are supplied by the CPU state. The register-number inputs to the decoder are supplied by the instruction, and they select one register within the current window. Some registers belong to two windows but have different numbers in each one of them; they are called "overlap registers". Other registers belong to a single window and are called "locals". The scheme works regardless of the numbering sequence in each window, as long as all windows have the same sequence. Figure 3.2.1 (a) shows a small register file with two overlapping windows. In addition, RISC I & II also have some registers, -- called "global" and not shown in fig. 3.2.1 -- which belong to all windows and have the same number in each.

The window number changes every time a procedure call is executed. Thus, every procedure activation record corresponds to a different window (overflows are dealt with in the next sub-section). The compiler allocates the local scalar variables of procedures into the "local" registers, so that no other activation record (window) has access to them. Thus, saving and restoring the registers on call and returns is not necessary. Local non-scalar variables, as well as scalar ones for which there are no registers available, are allocated on the execution stack in main memory, as usual.

The windows are organized in a stack. Parent and child procedure pairs are thus given adjacent, i.e. overlapping, windows. The compiler allocates the scalar arguments of procedures into the "overlap" registers. These registers appear with one fixed numbering to all parent procedures ("outgoing-argument"

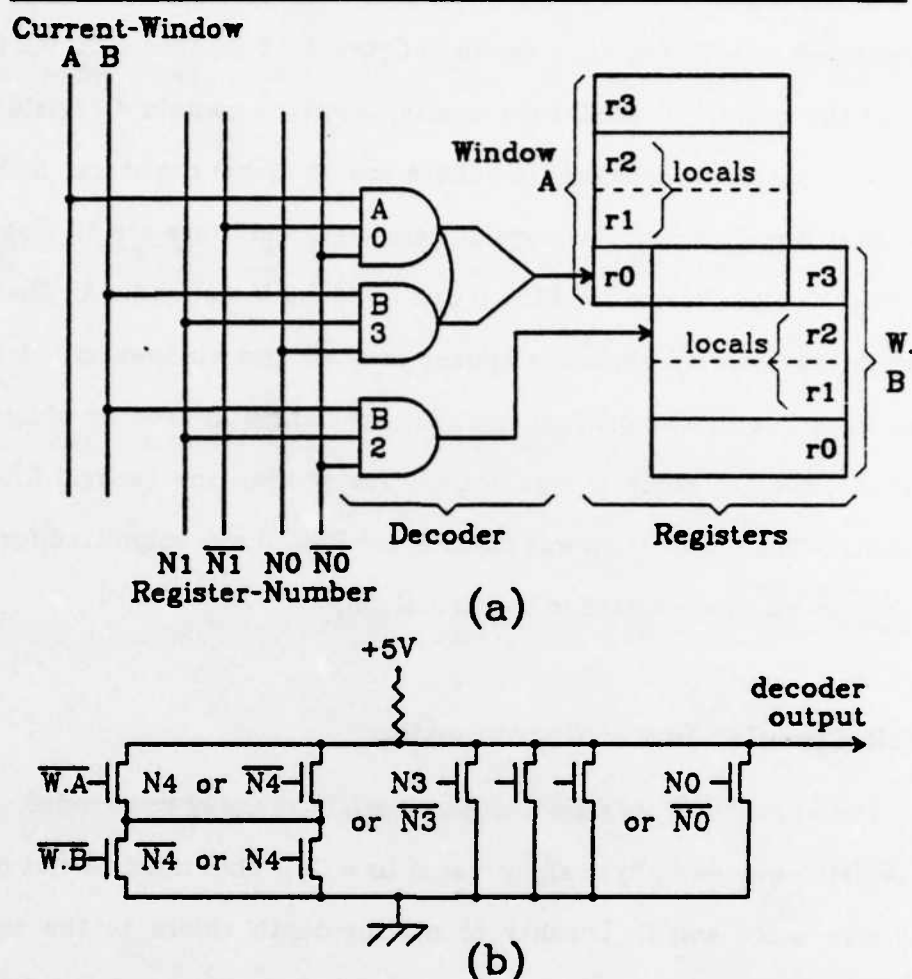


Figure 3.2.1: Overlapping Fixed-Size Windows.

registers), and with another fixed numbering to all child procedures ("incoming-argument" registers). In preparation for a procedure call, the parent writes the actual arguments into the former registers of the "current" window, and the child has them available in the latter registers of its own window. Thus, the overlap of windows allows for arguments to be passed in registers. These same "overlap" registers are also used for saving the return-PC, and for returning values from child to parent procedure.

In this scheme windows and overlaps have a fixed size, which allows the simple and fast AND-OR decoding shown in figure 3.2.1 (a) (see sect. 6.2 for a discussion of this point). In RISC I the overlap sections contain 4 registers, the local sections contain 6 registers, and there are 12 global registers. In RISC II overlaps have 6 registers, locals have 10 registers, and there are 10 global registers. The register numbering for RISC II can be found in Appendix A. That numbering is such that overlap registers appear in their two windows with 5-bit numbers that differ in only one bit position. The special NMOS decoder of figure 3.2.1 (b) is then possible, which is significantly faster than the general OR-AND-INVERT decoder. This observation was made *after* RISC II was submitted for fabrication, so the circuit was not used in the actual chip.

### 3.2.2 Circular-Buffer Organization.

The absolute procedure nesting depth is virtually unbounded. The number of register windows physically present in a CPU chip must be not only bounded but also quite small. Locality of nesting-depth refers to the relative depth changes of procedure nesting during a limited time interval, and implies that its fluctuations around a certain depth are fairly small. The CPU register windows are used for the few most recent activation records, for the top of the nesting stack. Older activation records may have to be saved in memory, when the nesting depth increases, and the windows which they occupy need to be re-used for younger procedures. Later on, as the depth decreases, these records have to be restored into the register file windows. The actual organization of the CPU windows is not an infinite stack, but rather a circular buffer for the top of that stack only. The rest of the stack is maintained in memory.

Figure 3.2.2 illustrates the circular-buffer organization. Two pointers are used to keep track of empty and occupied windows. The Current-Window-Pointer



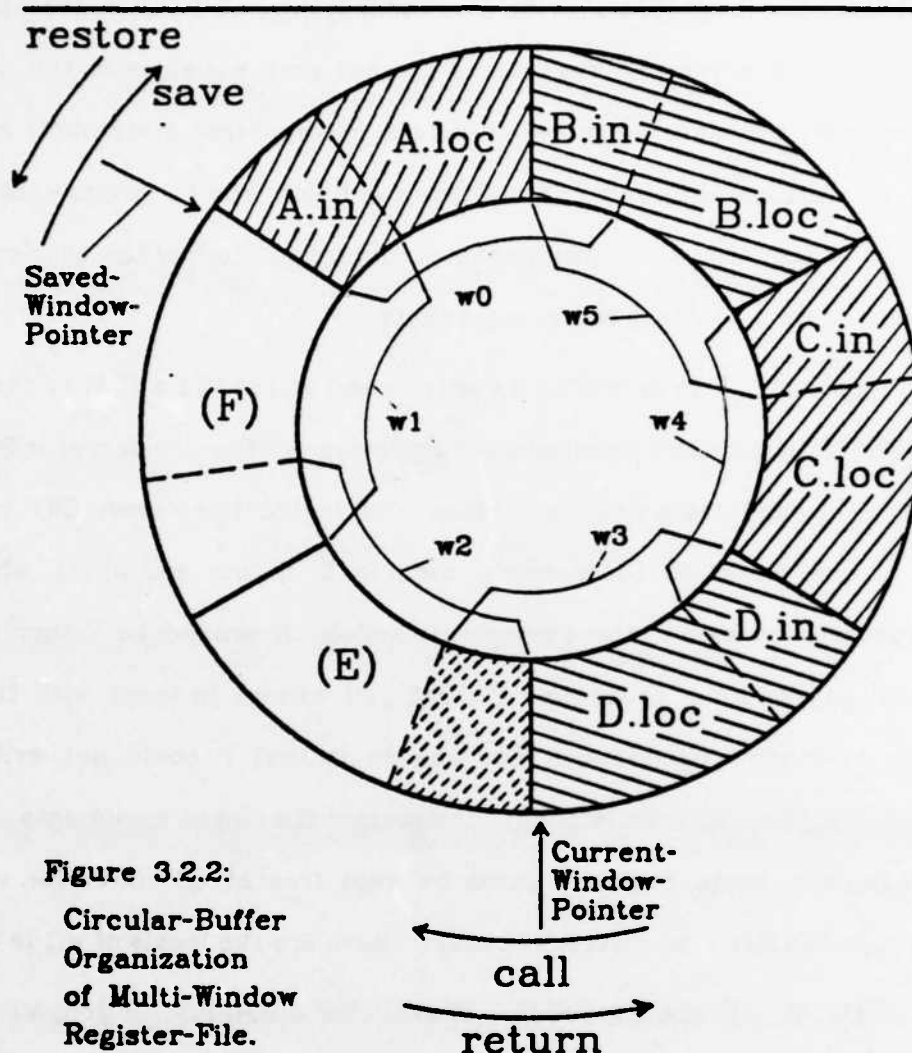


Figure 3.2.2:  
Circular-Buffer  
Organization  
of Multi-Window  
Register-File.

(CWP) points to the window of the currently active procedure ("window-number" in fig. 3.2.1). The Saved-Window-Pointer (SWP) identifies the youngest window that has been saved in memory. In the example of figure 3.2.2 a register file consisting of 6 windows is shown, with four of them being currently occupied. For grouping and identification purposes, the overlap registers are shown as belonging to that window in which they constitute input arguments. This grouping is important for the discussions that follow. The variables that are kept in overlap registers are only visible by the child procedure in the High-Level-Language. Only the child has a name for them and can reference them in

statements (in languages with up-level addressing, all further descendants can also do so). In contrast, these items are not even variables in the parent procedure, they are merely expression values, and no other statements in that procedure can reference them. This discussion holds true for arguments passed by value or value-return; for arguments passed by reference, the overlap registers actually contain pointers to the arguments.

If procedure D in figure 3.2.2 wants to call a procedure E, it writes the arguments of E in its outgoing-argument registers (in the overlap of w.3 with w.2), and then it executes a call instruction. Call instructions move CWP by one window in one direction (decrement, modulo 6, in our example), while return instructions move it in the opposite direction. If procedure E then decides to further call another procedure F, that call cannot proceed with the current status of window occupancy. The reason is that F could not write into its outgoing-argument area without destroying the input-arguments *A.in* of A. Furthermore, some registers must be kept free at all times for use by the interrupt-handler if an interrupt occurs; these are the locals of w.1 in our case.

At this point, when procedure E executes a call instruction, we say that a *register-file overflow* has occurred. A trap is generated, stopping the call instruction from completing execution. The criterion for the generation of this overflow trap is: *when a call instruction attempts to modify CWP so that it becomes equal to SWP*. The trap gives control to the overflow-handler routine, which saves one or more windows in memory. Tamir and Séquin have concluded that the best strategy, for most practical cases, is to save only one window per overflow trap [TaSe83]. In our example, the overflow-handler will save the areas marked *A.in* and *A.loc* in memory, i.e. only part of w.1, and will then appropriately move SWP to the start of *B.in*.

Similar considerations lead to the criterion for generating the underflow trap: *when a return instruction attempts to modify CWP so that it becomes equal to SWP*. Thus, a single equality comparator circuit is enough for detecting both over- and under- flows.

In summary, an  $N$ -window register file can hold only  $N-1$  activation records. (In the last table of § 2.2.2, figures were given for  $N-1$ ). Interrupts always modify the CWP in the same way as call's do. So, interrupt-handlers execute in a window where the local registers are guaranteed to be free. Interrupts should not be allowed to nest before the availability of more windows has been checked.

### 3.2.3 Pointers to Registers.

There are cases when a procedure's arguments or local scalars need to be accessed by a pointer or by one of its descendant procedures. The former is true in languages like C, where the programmer is allowed to ask for the address of a scalar variable and to use that address subsequently as a pointer for referencing the variable. That is for example the method for passing return arguments to procedures in C: `scanf("%d %d\n", &i, &j)`. The latter case, references to locals by descendant procedures, appears in languages with up-level addressing, like Pascal. It is usually implemented by maintaining a display of pointers to the bases of the activation records of the (static) ancestors. Thus, this amounts again to accessing a local variable via a pointer.

In this context, there are two methods to allocate local scalars to registers. The first one applies only to languages without up-level addressing. A two-pass compiler is used to recognize the variables which may have aliases and to allocate them in main memory. The second method is to provide means for correctly handling pointers to registers. The RISC architecture specifies the

latter approach. There has been a detailed design (data-path and timing) [Kate80] for the implementation of pointers to registers in the RISC I chip, resulting in no lost cycles. However, neither the RISC I nor the RISC II chips have implemented this scheme, because of lack of designer time. The RISC/E design (§ 1.2) does include the handling of pointers to registers. The RISC II micro-computer design contains off-chip circuitry to recognize the use of addresses pointing to registers and to generate an interrupt whenever that occurs [Liou83].

The proposed method for handling pointers to registers in a multi-window environment will be described here in a general form. It was developed in collaboration with D. Patterson in September 1980.

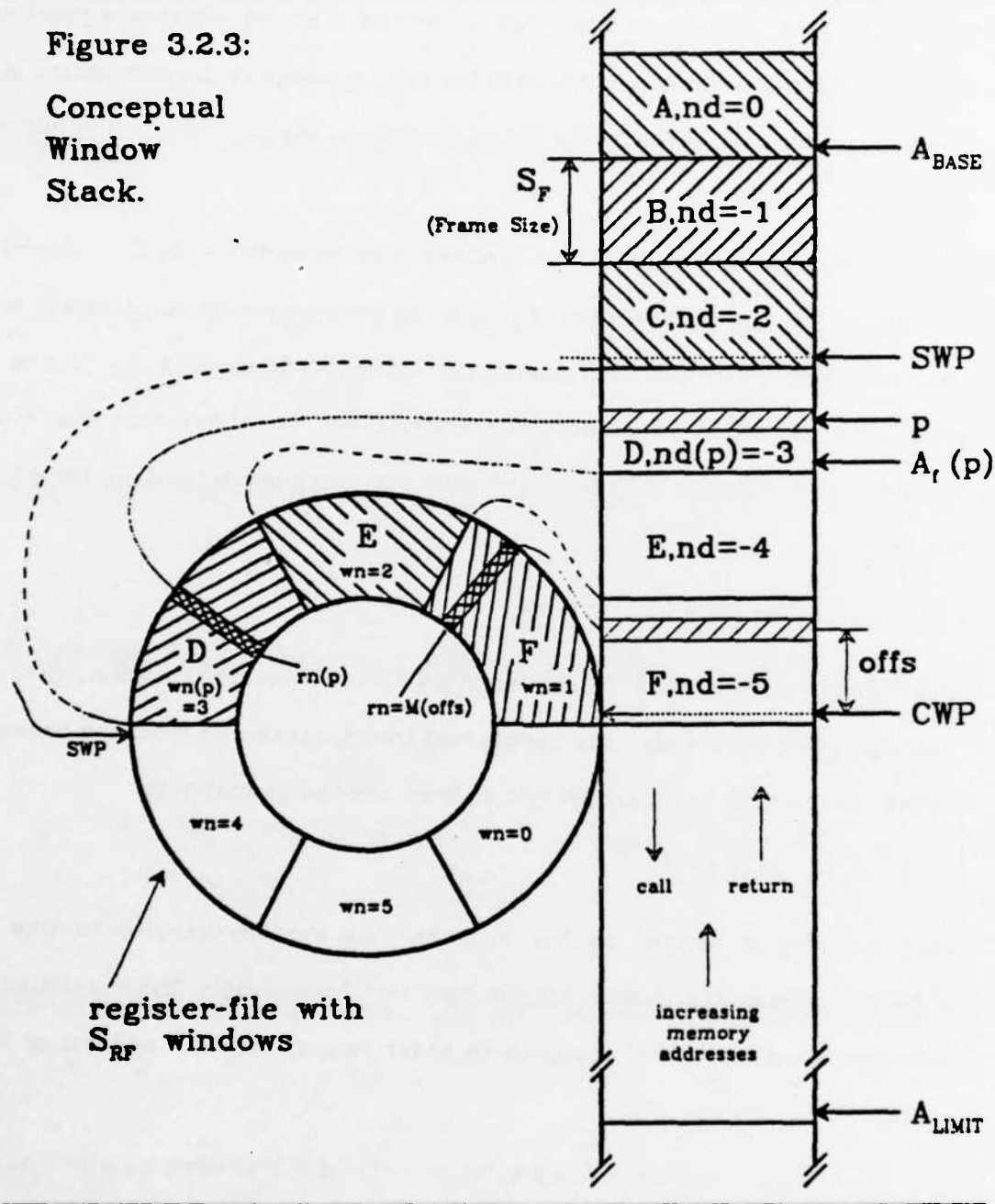
We use the notion of "*conceptual window stack*", a conceptual stack in memory consisting of a virtual image of the window frames of all active procedures. It contains one word of storage for each register of each window that has been "called" and did not yet "return". It is similar to the conventional execution stack of procedure activation records, except that RISC uses frames (records) of fixed size. Figure 3.2.3 illustrates this conceptual window stack. For clarity, window frames are shown without their overlap in that figure. The overlap registers are shown as belonging only to the window in which they constitute input arguments, according to the discussion in § 3.2.2. Thus, the "frame size",  $S_F$ , shown in that figure is equal to the number of incoming-argument and local registers in one CPU register window, which is also the amount by which the window pointer is moved on each call or return.

All but the top of the conceptual window stack is actually present in memory in the form of older windows saved there, as discussed in § 3.2.2. The top of that stack is in the registers on the CPU. Figure 3.2.3 illustrates this arrangement. The address of a register, at a given time, is the address of the

word of the conceptual stack, which it is holding at that particular time.

Figure 3.2.3:

Conceptual  
Window  
Stack.



In what follows, the window stack is assumed to grow towards decreasing addresses. Window frames are uniquely identified by their nesting-depth  $nd$ , which is a negative number. That is related to the frame address  $A_f$ :

$$A_f = A_{BASE} + nd \cdot S_F \quad (1)$$

where  $S_F$  is the size of the window-frame as defined above. The Current-Window-Pointer ( $CWP$ ) and Saved-Window-Pointer ( $SWP$ ), which were introduced in the last subsection, are generalized here to be the memory addresses pointing to the same frames as before, but now in the virtual image in the conceptual stack. They thus define the boundaries of the portion of that stack which is currently kept in the register file.

The register file is of size  $S_{RF}$  windows, numbered  $wn = 0, 1, \dots, S_{RF}-1$ . By convention, the procedure with  $nd=0$  is given the window  $wn=0$ . Also by convention, each procedure call decrements the current  $wn$ , modulo  $S_{RF}$ . Since each procedure call also decrements the current  $nd$ , it follows that the window-number  $wn$  of a frame with nesting-depth  $nd$ , which is currently in the register file, is:

$$wn = nd \bmod S_{RF} \quad (2)$$

The register-number  $rn$  of a register in a CPU window, and the offset,  $offs$ , of the corresponding word within the conceptual frame, measured from the base of the frame, are related by an arbitrary but fixed one-to-one mapping:

$$rn = M(offs) \quad (3)$$

That mapping is defined by the way that the overflow-handler routine saves registers in memory frames, and it may well be a simple linear relation. The compiler must know that mapping in order to generate the address of a local scalar.

The memory address of an argument or local scalar variable which has been allocated into register  $rn$  can thus be computed by:

$$CWP + M^{-1}(rn) \quad (4)$$

where  $CWP$  is known at run-time when the procedure is entered.



Now assume that a memory reference is made using a pointer  $p$  as effective address. Special action has to be taken if and only if  $p$  is pointing to a register:

$$CWP \leq p < SNP \quad (5)$$

If that last condition holds true, then the window-number  $wn(p)$  and the register-number  $rn(p)$  of the register where  $p$  is pointing to must be determined, so that the memory reference can be correctly turned into a register reference. Let  $A_f(p)$  be the base address of the conceptual frame where  $p$  is pointing to. Since  $0 \leq p - A_f(p) < S_F$ , and since  $A_f(p) - A_{BASE}$  is a multiple of  $S_F$  (by equation (1)), it follows that:

$$\left\lfloor \frac{(p - A_f(p)) + (A_f(p) - A_{BASE})}{S_F} \right\rfloor = \frac{A_f(p) - A_{BASE}}{S_F} \quad (6)$$

Combining this with equation (1), we get:

$$nd(p) = \left\lfloor \frac{p - A_{BASE}}{S_F} \right\rfloor \quad (7)$$

and combining with equation (2) we find the window-number where  $p$  is pointing to:

$$wn(p) = \left\lfloor \frac{p - A_{BASE}}{S_F} \right\rfloor \bmod S_{RF} \quad (8)$$

Finally, to get the register-number  $rn(p)$ , we will use equation (3) and the property  $a \bmod b = a - \lfloor a/b \rfloor \cdot b$ :

$$\begin{aligned} rn(p) &= M(offs(p)) = M(p - A_f(p)) = \\ &= M(p - A_{BASE} - nd(p) \cdot S_F) = \\ &= M\left(p - A_{BASE} - \left\lfloor \frac{p - A_{BASE}}{S_F} \right\rfloor \cdot S_F\right) = \\ &= M((p - A_{BASE}) \bmod S_F) \end{aligned} \quad (9)$$

All this complicated arithmetic reduces to trivial bit-field extractions and concatenations, when the pertinent constants are powers of 2. Such is the case in RISC II:

$$\begin{aligned} \text{RISC II: } A_{BASE} &= 2^{32} \\ S_F &= 64 \text{ (bytes)} \\ S_{RP} &= 8 \\ M(\text{offs}) &= 16 + \text{offs}/4 \text{ (byte addresses).} \end{aligned}$$

Under these circumstances, the important equations become:

$$(\text{Address of local or input-arg. in } R_n) = CWP + (n-16) \cdot 4 \quad (4')$$

$$wn(p) = p \langle 8:6 \rangle \quad (8')$$

$$rn(p) = 1 \# p \langle 5:2 \rangle \quad (9')$$

where  $F \langle m:n \rangle$  is the bit-field extraction operator, and  $F_1 \# F_2$  is the bit-field concatenation operator.

Concerning the detection of pointers  $p$  addressing registers, equation (5) says that two full-address comparisons are needed. The comparison of  $p$  with  $SWP$  is required in order to decide whether  $p$ 's frame is currently in a register file window or has been saved in memory. The comparison of  $p$  with  $CWP$  is required in order to decide whether  $p$  is pointing into the conceptual window stack or simply to something else in memory. However, this latter condition can also be checked by comparing  $p$  against  $A_{LIMIT}$ :

$$A_{LIMIT} \leq p < SWP \quad (5')$$

where  $A_{LIMIT}$  is the boundary address of the portion of virtual memory allocated for the window stack (see bottom of fig. 3.2.3). If  $A_{LIMIT}$  is a "convenient" hardwired constant, then the comparison of  $p$  against it may be implemented with very simple hardware. In RISC II,  $A_{LIMIT} = 2^{32} - 2^{24}$ , and that comparison reduces to  $p \langle 31:24 \rangle = 11111111$ , which can be checked with a single NOR gate.

If the window size or register-file size are not powers of 2, then the proposed method is to modify the definitions of  $S_F$  and of  $nd$  in the following way:  $S_F$  should be defined as the smallest power of 2 that is large enough for a frame to fit in. The  $nd$  counter, which counts down/up on every call/return, should be made into a conventional counter for its most-significant bits, coupled with a modulo- $S_{RF}$  counter in its least-significant bits. This will waste some words in main memory, where the window stack is kept, but this solution is preferable to implementing hardware to carry out integer divisions by arbitrary constants.

### 3.3 The RISC I & II Pipelines.

The pipeline organizations of RISC I and II are presented here. They form the basis of the micro-architecture of these two implementations, and they have even influenced the original definition of the RISC I & II architecture. RISC I has a two-stage pipeline, while RISC II has three stages. The issue of pipeline suspension during data memory accesses is discussed. Other possible pipeline schemes are reviewed. The default presence of an addition in all register-to-register moves and in all addressing modes of RISC is explained.

#### 3.3.1 Two and Three Stage Pipelines.

Most RISC instructions (sect. 3.1) can be executed within the same amount of time, adhering to the following execution pattern:

- read  $R_{s1}$  and  $R_{s2}$  (or get  $PC$  or  $imm$ ),
- perform an add/sub or logic or shift operation on  $S1$  and  $S2$ , and

- write the result into  $R_d$ , or use it as an *effective-address* for a memory access.

Load and store, the only instructions containing a data memory access, require an additional cycle for completing their execution. They will be discussed in § 3.3.2. The simple execution pattern of the RISC instructions leads to simple pipeline schemes. Figure 3.3.1 shows the RISC I and II pipelines, along with the resulting utilization of the data-path.

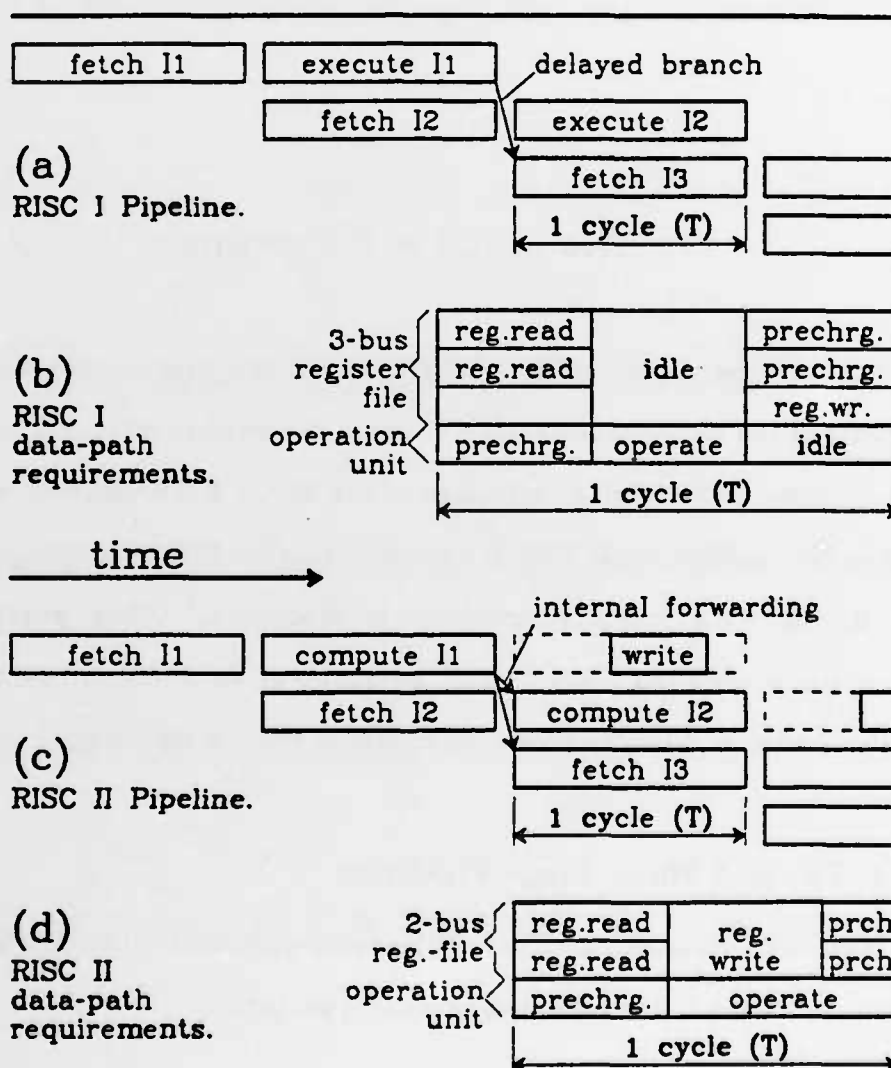


Figure 3.3.1: The RISC I and II Pipelines.

RISC I (fig. 3.3.1(a)) has a simple two-stage pipeline, overlapping instruction fetch and execution, and including the delayed-branch scheme (section 3.1.3). It is assumed that an instruction-fetch memory cycle takes roughly the same amount of time as a CPU read-operate-write cycle. Figure 3.3.1(b) shows the requirements placed on its data-path. For a good performance, a two-port register read is specified, in order to simultaneously get  $R_{s1}$  and  $R_{s2}$ . Next, the operation is performed onto the two sources, while the register file remains idle. After that one has completed, the result can be written into  $R_d$ , while the operational unit, now, remains idle. For an efficient NMOS implementation the two read-busses of the register-file have to be precharged before a read operation is made. In order to reduce the cycle time, RISC I precharges those busses in parallel with writing  $R_d$ . Thus, its register file must have 3 busses.

In RISC II a third pipeline stage was introduced (fig. 3.3.1(c)), and the writing of  $R_d$  was delayed until that stage. Internal forwarding is used to resolve register interdependencies among subsequent instructions in the pipeline: Two equality comparators detect the conditions  $R_{s1J2} = R_{dJ1}$  or  $R_{s2J2} = R_{dJ1}$ . When these occur, the result of  $J1$ 's operation is automatically forwarded from the temporary latch where it is kept, for use by  $J2$ , in lieu of the stale contents of  $R_{dJ1}$ .

The requirements that this pipeline scheme places on the data-path are radically different from the previous ones (figure 3.3.1(d)). Here, the register file is kept busy all the time. It performs the write of  $R_{dJ1}$  immediately after the reads of  $R_{s1J2}$ ,  $R_{s2J2}$ . The register-write operation and the precharging of the register-file busses are done in parallel with the ALU or shift operation, instead of sequentially after it as the two-stage pipeline requires. This results in a performance gain, part of which can be spent to perform the precharging *after* the register-writing, thus allowing the use of a two-bus register file. The

more compact two-bus register-cell is the main reason why the RISC II chip could pack 75% more registers than RISC I into a 25% smaller area. If an ALU or shift operation takes as much time as a register-write *plus* precharging the register file busses, then precharging the register file busses after the write operation will result in *no* performance loss relative to precharging and writing in parallel, with a three-bus scheme.

### 3.3.2 Pipeline Suspension during Data Memory Accesses.

The RISC I and II CPU chips have a single memory port and assume a non-pipelined memory. This means that only one memory access may be in progress at any time. As a result, when the data memory access of a load or store instruction is being carried out, the rest of the pipeline is temporarily suspended, because an instruction-fetch access cannot be processed at the same time. This situation is illustrated in figure 3.3.2 (a).

The limitation of a single memory port is quite common in microcomputer systems. It is the result of pin constraints, of the presence of a single, non-pipelined memory bank, and of the absence of on-chip cache(s). However, as § 6.3 suggests, it is desirable to integrate an instruction cache on a RISC-style CPU chip, once the technology makes that feasible. An on-chip cache appears as an independent memory port to the CPU, whenever a miss does not occur. The CPU then effectively sees two separate memory ports, one for instructions and one for data. Thus, it is appropriate to study pipelines which are not suspended on data memory references, as figure 3.3.2 (b) shows.

When the constraint of single memory access per cycle is removed, the data access cycle of load and store instructions can occur in parallel with the compute cycle of the next instruction. For store instructions, this poses no problem of data dependency. For load instructions, however, it does. The compute cycle



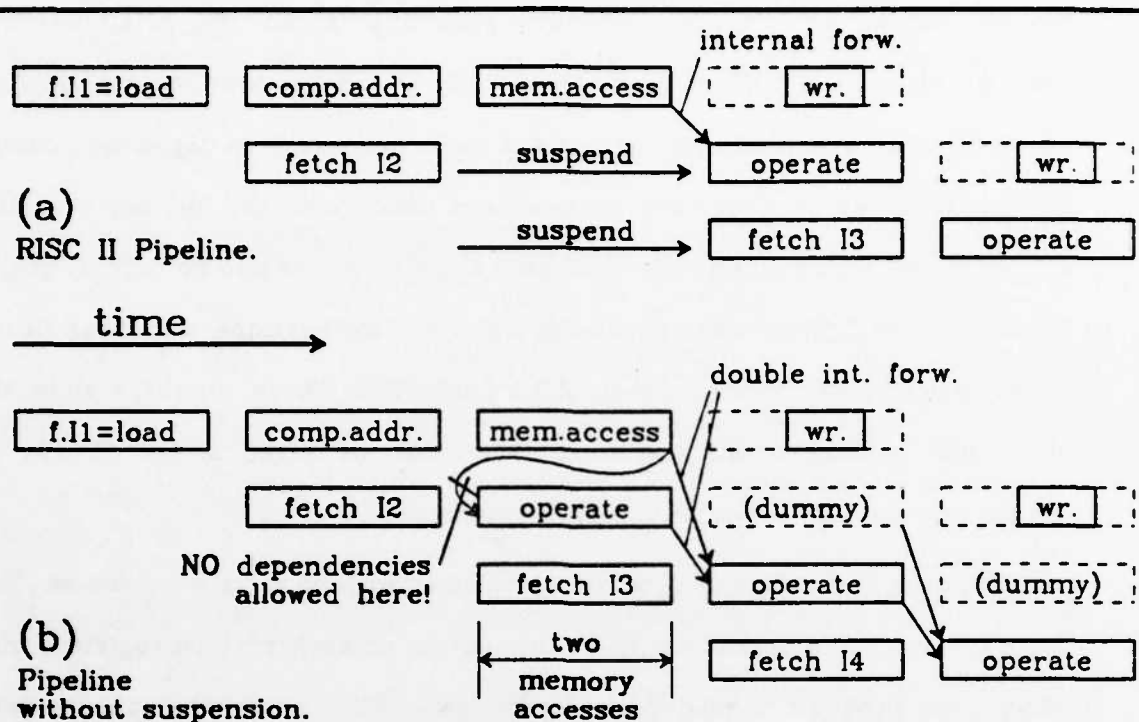


Figure 3.3.2: Pipeline Suspension during Data Memory Accesses.

of the instruction immediately following a load must not depend on the value being loaded. This condition needs to be checked by the compiler, which may insert a no-op if no useful work can be done in the slot after the load instruction. Alternatively, such a dependency may be detected by hardware, which then suspends the pipeline while waiting for the data to arrive from memory.

If the register-file can only handle a single register-write per cycle, as in the case of RISC II, a dummy pipeline stage has to be inserted into *all* instructions at the place where loads perform their memory access (figure 3.3.2 (b)).

An advanced pipeline scheme using dual memory write-ports certainly speeds up load and store instructions. But what is their overall impact on performance? How frequent are store instructions, and how frequent are load instructions followed by a computation that doesn't depend on them? John

Cocke of the IBM Watson Research Center gave the following numbers regarding the 801 during an informal discussion [Cock83]. About 16% of all executed instructions (on the 801) are loads followed by an independent computation, and about 9% of all executed instructions are loads followed by a dependent computation. As far as data memory accesses are concerned, the 801 has a pipeline similar to fig. 3.3.2 (b), but with dual-port register writes and no dummy stages. Cocke gave no figure on the percentage of store instructions, but these usually range around 10% (see e.g. sect. 2.2.1 [AlWo75]). These numbers show that about one quarter of all execution cycles can be saved in the 801 by not suspending the pipeline on data memory accesses.

However, these figures concern a processor with *no register windows*. Such processors need to access variables in memory or save/restore registers more often than processors with register windows. RISC programs execute fewer loads/stores. In three measured program runs, 17%, 13%, and 15%, respectively, of all executed instructions were load's [PaSe81, fig.15]. The corresponding percentages for store instructions were 1%, 1%, and 9%. In RISC, the instructions following the load's can be expected to depend on them more frequently than in other architectures, because restoring multiple registers from memory, upon procedure returns, is much less frequent. Thus, the percentage of RISC execution cycles that can be saved by allowing simultaneous instruction-fetches and data-memory-accesses, can be estimated to be in the range of 10%.

During the RISC design process, another possibility was considered. If no other instruction can be fetched for execution during the data-memory-access cycle, one could try to pack more information into the load/store instruction itself, so that the CPU can do something useful during the above cycle. As an example, a third instruction format could be introduced, where the short-SOURCE2 of figure 3.1.1 would be split into a 9-bit source-2,  $S_2$ , and a 5-bit  $R_{s3}$

specifier. Load and store instructions having this format would perform the following operations during their two execute-cycles:

- $eff\_addr \leftarrow R_{s1} + S2$
- $R_d \leftarrow M[eff\_addr]$ ; compare-&-set-CC's:  $R_{s1}-R_{s3}$

These instructions could be used for implementing combinations of HLL statements such as:

`c = *p ; if ( p >= limit ) ....`

Such combinations are quite rare, however. For example, in the critical loops of section 2.4, there are some program segments in procedure *gline()* of *sed* (2.4.2) that come close to the above; however, still none of them is suitable for this optimization. In any case, the inclusion of instructions like the above would lengthen the basic processor cycle-time, because additional register-number latches and multiplexors would be required in the critical path of register-number decoding (section 4.2). Thus, such instructions were not included in the RISC architecture.

### 3.3.3 Other Pipeline Schemes, and the Issue of Default Addition.

More pipelining than what RISC II has is possible in RISC-like register-to-register architectures. Figure 3.3.3 compares the 3-stage RISC II pipeline (a), with the 4-stage 801 pipeline (b) [Cock83]. The 4-stage pipeline pushes the data-path utilization as far as data-dependencies permit. The result of an arithmetic, logic, or shift operation may be used as a source for the operation of the next instruction as soon as it becomes available (arrow (1) in fig. 3.3.3(b)). In order to avoid doubly-delayed jumps (arrow (2) in the same figure), the 801 performs the addition of the *PC* with the immediate offset, for PC-relative branches, in parallel with the source-register reads (arrow (3)). Of course, the 4-stage pipeline places heavier requirements on the register file and on the instruction-

fetch mechanism. Register reads and writes are performed in the same cycle, and the time to fetch an instruction must be as short as the time to perform an addition. All these issues are studied in more detail and to a greater depth in Robert Sherburne's thesis [Sher83].

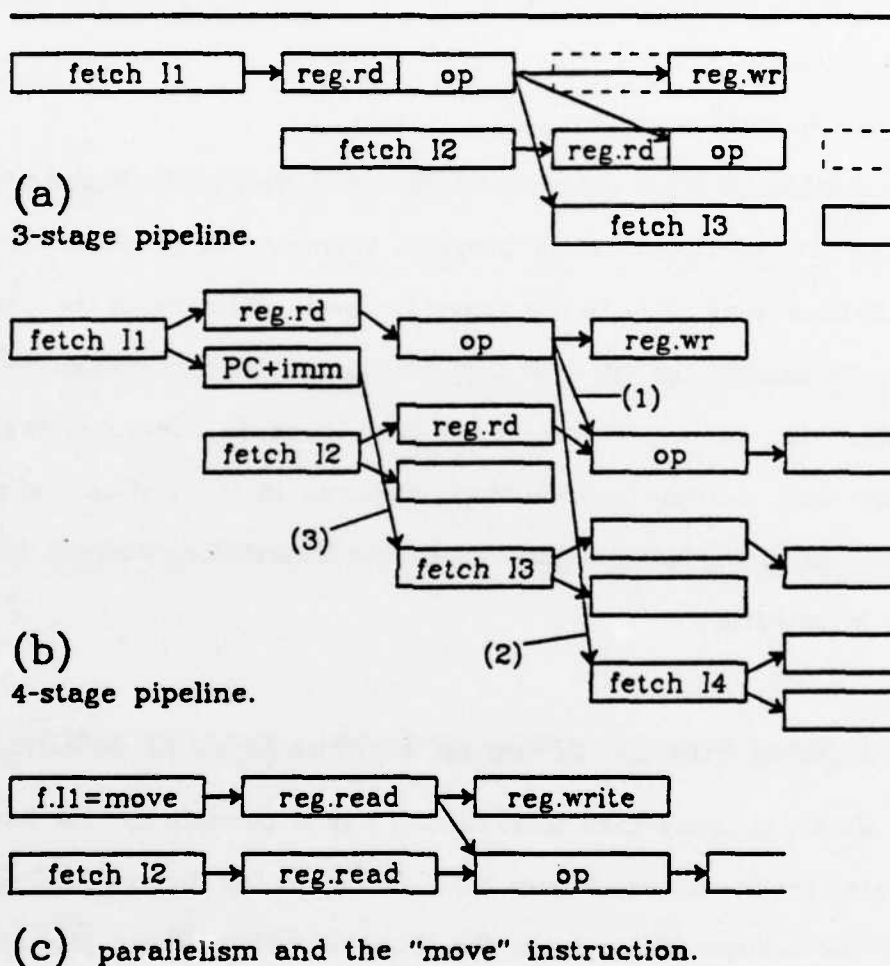


Figure 3.3.3: Various Pipelines.

In the Berkeley RISC architecture there is no register-to-register *move* instruction. It is synthesized by executing  $R_d \leftarrow R_s + 0$ , thus performing a dummy ALU or shift operation which by default exists in every RISC instruction. This architectural decision can be explained on the basis of the pipeline organization. In RISC I (fig. 3.3.1(b)), *move* instructions with no ALU/shift operation

could execute about 30% faster. However, the corresponding shorter cycles would make timing irregular and introduce significant implementation difficulties. Furthermore, they would require a 30% faster instruction-fetch mechanism, even though this increase in speed could not be exploited during the rest of the cycles. Referring to figure 3.3.3, it can be seen that, in RISC II (a), and in the 4-stage pipeline (b), removing the *op* part from one instruction execution, would yield no performance gain either, as long as the pipeline is limited by instruction-fetches and by register-file accesses. In part (c) of the figure, a pipeline is shown which could exploit the available parallelism in the case of a move instruction. Two instructions would have to be fetched and executed simultaneously. The MIPS processor [Henn83] does allow two instructions to be packed in one word and be fetched from memory simultaneously. However, each major execution cycle (pipeline-step) contains two minor cycles. When two register-to-register instructions are packed together, they are in fact executed sequentially -- each during one minor cycle -- because of the lack of sufficient hardware resources to support more parallelism.

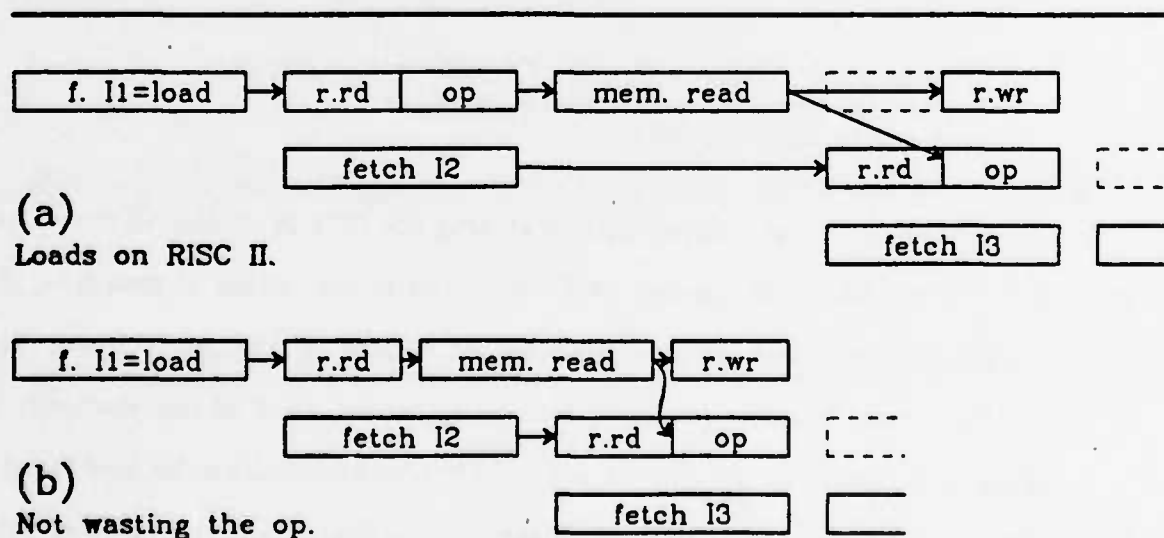


Figure 3.3.4: The issue of Default Addition.

Another related issue is the single addressing mode of RISC I & II, which always performs an addition when computing the effective-address, regardless of whether it is needed or not. References such as mere pointer indirections ( $*p$ ) are executed by  $R_d \leftarrow M[R_p + 0]$ , and the addition is wasted. The reasons for this architectural decision again have to do with the pipeline scheme and with the single memory-port. Figure 3.3.4 (a) shows the execution of a load instruction in the RISC II pipeline. If its address calculation requires no addition, then the data-memory-read operation could be performed half a cycle earlier, as shown in part (b). This would allow the next instruction to start executing one half or even one full cycle earlier. However, the data memory access would have to overlap with the instruction fetch accesses, something which RISC I & II cannot do, as discussed in § 3.3.2. Section 6.4.1 will show how the timing of fig. 3.3.4(b) is possible *including* the addressing addition if a data-cache is used.

Instead of trying to start the memory access and the next instruction earlier than normal, some other useful work could be done in lieu of the unnecessary address addition. Examples of what these modified load/store instructions could do are:

- choice of:  $eff\_addr = R_{s1}$ , or  $eff\_addr = R_{s1} + S2$ , and
- optionally compare-&-set-CC's:  $R_{s1} \pm S2$ , or
- optionally:  $R_{s1} \leftarrow R_{s1} + S2$ .

The former option of comparing and setting the CC's is similar to the variation that was examined at the end of § 3.3.2. The latter option of modifying  $R_{s1}$  is similar to the auto-increment/decrement modes of the PDP-11 and VAX-11 architectures. Shustek [Shus78] found that about 15 % of the statically used addressing modes on the PDP-11 are auto-inc/dec modes, but he also found that some of them were merely used to increment/decrement the register without using the accessed memory word. The auto-inc/dec modes are well suited for



stack accesses, which the PDP-11 uses a lot because it has few registers (only 32% register-mode usage, in the above study). RISC, on the contrary, performs many fewer memory references, thus the gain which auto-inc/dec mode could bring is limited. Besides, a complication would arise with these modes. Writing into  $R_{s1}$  would have to occur during the data-memory-access and if that access leads to a page-fault interrupt,  $R_{s1}$  would already have been modified. We preferred not to implement any of those modified load/store instructions, in order to stay with a clean and simple architecture.

### 3.4 Evaluation of the RISC I & II Instruction Set.

In this section we evaluate the Berkeley RISC architecture from various points of view. First, its most controversial part, the reduced instruction set is considered. We discuss its appropriateness for a High-Level-Language (HLL) computer, its impact on code size, and its effect on machine performance. The overall machine is then evaluated, taking into consideration the large multi-window register file, the reduced design time, and the elimination of design errors.

#### 3.4.1 Instruction Set and High-Level-Languages.

RISC instructions have some similarity to the micro-instructions on typical micro-programmed machines, some of which will be further discussed in chapter 4:

- All instructions have the same width, and most of their fields have fixed size and position (3.1.4).
- All instructions execute in the same amount of time (except for the "minor" irregularity of pipeline suspension during loads/stores) (3.3).
- All instructions follow a similar and fixed pattern of execution in the data-path (4.1).
- Delayed branches are used (3.1.3).
- The instruction decoder is so simple that it occupies only 0.5% of the chip area (4.4).
- The control signals which sequence the execution of instructions in the data-path are generated by some simple gates that occupy just 1% of the chip area (4.4).

Based on these similarities, some people argue that the RISC instruction set is "of too low a level for a High-Level-Language computer".

However, several frequent HLL statements are compiled into only a single or a few RISC machine instructions. Here are some examples from the critical loop of *fgrep* (§ 2.4.1):

<u>HLL statement:</u>	<u>RISC machine instructions:</u>
if (--ccount <= 0)	sub-&-set-CC's: $R_{ccount} \leftarrow R_{ccount} - 1$ jump-if-less-or-equal
c->inp == *p	load: $R_{t1} \leftarrow M[R_c + OFFS_{inp}]$
(from ccomp() macro)	load: $R_{t2} \leftarrow M[R_p + 0]$
	sub-&-set-CC's: $R_0 \leftarrow R_{t1} - R_{t2}$
c = c->fail	load: $R_c \leftarrow M[R_c + OFFS_{fail}]$

Thus, RISC machine instructions are not far away from some very frequent HLL statements. We could even argue here that the variants of the load/store instructions which we examined at the end of section 3.3.2 and 3.3.3 actually correspond to *two* HLL statements each.

The topic of High-Level-Language computers (HLLC) has attracted much interest among computer architects and programmers during the last two decades. Some view a HLLC as a machine that should reduce the "semantic gap" between HLL's and machine code. However, Ditzel and Patterson argue that there is no obvious justification as to why this should be a desirable goal

[DiPa80]. Instead, they define a HLLC as one where all programming, debugging, and error reporting takes place in a HLL, so that the user need not be aware of the existence of the machine language. Thus, whether an instruction set is close to micro-code or close to HLL statements is irrelevant to the issue of HLLC. What is important is whether a compiler and a symbolic debugger can be built for a particular architecture, and how fast compiled HLL programs run.

Writing compilers for RISC has proven quite easy, because the instruction set provides simple and straightforward primitives for synthesizing HLL functions. Johnson's Portable C Compiler (PCC) and a peephole optimizer have been modified in less than 6 person-months, to produce code for RISC [Camp80]. Miros also produced another, more solid, C compiler for RISC, again modifying the PCC [Miro82]. This RISC PCC had one third less code-table entries than the comparable VAX-11 PCC.

Other measures can be used to show that RISC is no less a High-Level-Language architecture than are other favorite processors. Campbell [Camp80] gives the static number of machine instructions in 12 C programs, compiled and optimized for the RISC, for the VAX-11, and for the PDP-11. Relative to the VAX-11 code, the PDP-11 code has 40% more instructions, and the RISC code has 67% more instructions, on the average. This shows that, although RISC instructions do contain "less information" than VAX or PDP instructions and could thus be considered "lower level", the difference is not at all that dramatic.

Figure 3.4.1 contains some performance measurements of 5 programs with no procedure calls, published in [PaPi82] and adjusted here for the measured 330nsec and 500nsec cycle times of two scaled versions of RISC II chips (§ 5.2). Execution times of C and Assembly versions of 5 programs are given, normalized relative to the C version on a 500nsec RISC II. We will come back to these performance measurements in § 3.4.3. Of interest here is the ratio of the assembly-

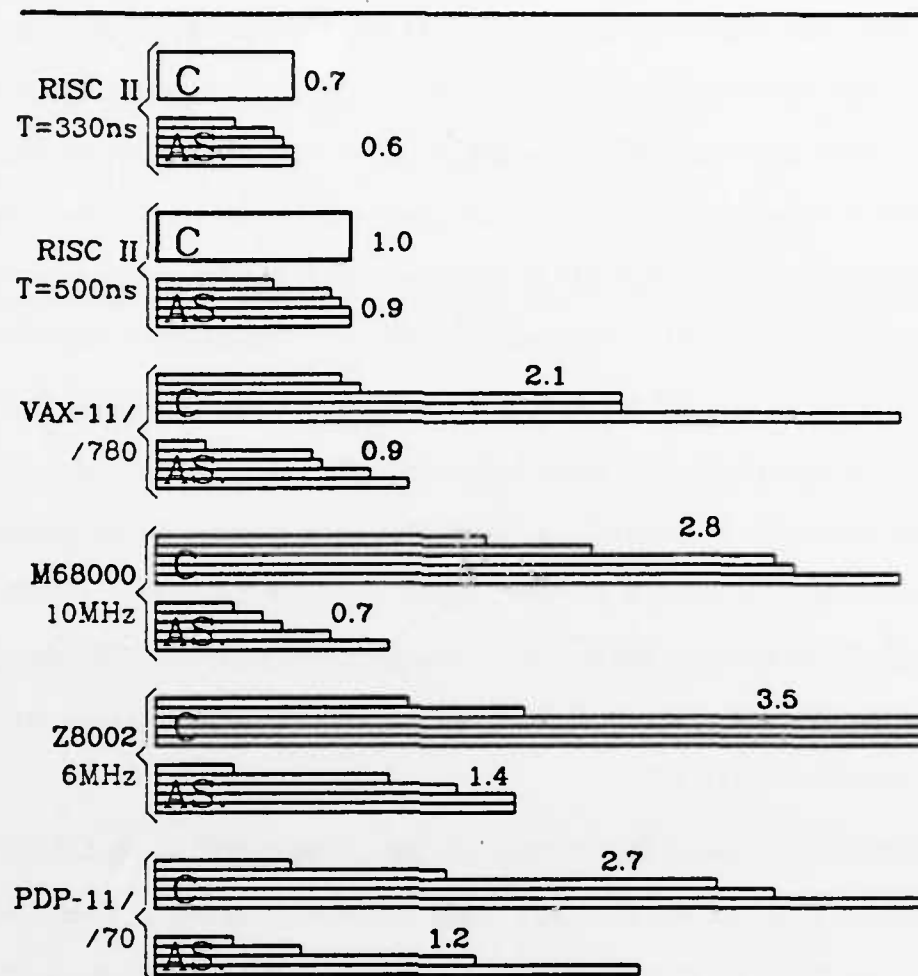


Figure 3.4.1: Normalized execution time of 5 EDN benchmarks (without procedures), on 5 machines, in C and in Assembly.

code execution-time to the compiled-code execution-time on the various machines. The averages of the corresponding ratios are as follows:

Machine:	<i>average</i> ( $\frac{\text{Assembly-Code Execution-Time}}{\text{Compiled-Code Execution-Time}}$ )
RISC	0.90 $\pm$ 0.1
PDP-11/70	0.50 $\pm$ 0.2
Z8002	0.46 $\pm$ 0.3
VAX-11/780	0.45 $\pm$ 0.2
M68000	0.34 $\pm$ 0.3

This ratio is a measure of the loss in performance due to programming in a HLL rather than in assembly language. The lower this ratio, the more the programmer is tempted to write assembly code. Using this measure, RISC is the best HLL architecture among the ones examined above [PaPi82]. It can be seen that compilers have difficulties to make effective use of the complex instructions that other processors provide.

### 3.4.2 Instruction Set and Code Compactness.

An instruction set and an instruction encoding that achieve compact code are desirable for two main reasons. Firstly, they allow the computer system to have smaller memory devices for holding the same amount of compiled programs. Memory devices, here, are disks, main memory, and (instruction) cache. By being smaller, these devices can be faster and cheaper. Or, alternatively, memory devices of the same size can hold more compiled code. Secondly, when the machine code is more compact, less bandwidth is necessary for fetching instructions into the CPU at the desired rate. This allows busses to be cheaper. Alternatively, the same bandwidth will allow faster fetching -- and thus faster execution -- of the compiled program.

However, in several actual situations, the above effects may be weak, while achieving code compactness may be expensive in other ways. There are two main methods for reducing the average code size. Firstly, an instruction format

closer to Huffman encoding may be utilized. This means having a variable number of fields in the instructions, and possibly having the fields encoded with variable sizes. The choices are made according to the relative frequency of usage of instruction and field types. Secondly, frequent sequences of related primitive operations can be made into single instructions. This allows the elimination of fields specifying intermediate results, or of multiple fields specifying common operands. It reduces the number of instructions that need to be fetched.

Instruction encoding and combining must be done carefully, to avoid some possible negative effects on CPU performance and cost. The circuitry that decodes instructions and controls their execution can become large and costly if the instruction encoding is too complicated. Performance can be severely impaired if decoding the instruction must involve *serial* rather than parallel operations. The extraction and interpretation of critical instruction fields should not depend on previous complex decodings of other fields. Also, instructions that require a long execution time, with many intermediate results, may necessitate the inclusion of too many latches into the data-path. This may slow down execution, due to increased capacitive loading, and may render interrupt handling awkward and slow.

Trying to improve performance by compacting the machine code, in order to alleviate the instruction-fetch bottleneck, has its limitations. Firstly, instruction-fetch is usually overlapped with instruction execution, and thus, reducing the instruction-fetch time beyond the available overlap brings no performance gain. Secondly, unless a sophisticated buffering mechanism is used, fetching an instruction takes an amount of time equal to

$$\left\lceil \frac{\text{Instruction Width}}{\text{Bus Width}} \right\rceil \cdot (\text{Bus Cycle Time})$$

In other words, instruction pieces that are narrower than the bus width still require a full cycle to be fetched. Furthermore, instructions of integer word-width may require an additional fetch cycle if they are not aligned on word boundaries. The cost of an instruction-buffering mechanism that could remedy such problems is rarely lower than the cost of simply increasing the width of the bus and of the memory devices, and thus achieving the same fetch rate with wider instructions that are more conveniently aligned.

All these considerations convinced us to stay with the simple instruction set, and with the two fixed and regular instruction formats of section 3.1, even though they are somewhat wasteful in code size. Instructions are word-aligned, and their width is always one word. Thus, exactly one cycle - the minimum possible - is required to fetch any instruction. The execute-cycle of instructions was defined to perform as much work as practically possible during the one cycle that it takes to fetch the next instruction. As a result of the simple instruction format, the decoding and field-extraction circuitry is trivial, at most 1 or 2 % of the chip area (§ 4.3). The relevant trade-offs were studied carefully, such as long constants (§ 3.1.4) and modified loads/stores (§ 3.3.2 and § 3.3.3). However, in all cases, the simpler solution looked better. After all, memory costs are decreasing, and "wasting" memory is quite common. For example, a full word (32 bits) is usually allocated for every integer, regardless of its actual range.

Even though RISC has such a simple instruction set and instruction format, its average code size is only modestly larger than that of other processors:



<i>Code Size Relative to RISC</i>		
Machine:	Averaged over:	
	11 C programs [PaSe82]	12 C programs [Camp80]
RISC I, II	1.0	1.0
VAX-11/780	0.8 $\pm$ 0.3	0.67 $\pm$ 0.05
M68000	0.9 $\pm$ 0.2	
Z8002	1.2 $\pm$ 0.6	
PDP-11/70	0.9 $\pm$ 0.4	0.71 $\pm$ 0.12
BBN C/70	0.7 $\pm$ 0.2	

We see that RISC code is usually not more than 50% larger than the rather compact VAX-11 code.

Garrison and VanDyke have studied how much RISC code size could be reduced by encoding the same instruction set with variable-length fields and instructions [GaVD81]. Their results, which are also reported in [Patt83], indicated that the following savings, relative to the present RISC format, are possible:

Huffman encoding (4 to 67 bits/instr.)	43 % savings
8-, 16-, 24-, and 32- bit instructions	35 % savings
16- and 32- bit instructions	30 % savings

The last encoding is done by introducing half-word encodings for 7 special cases of existing RISC instructions. This simple encoding certainly brings RISC code size into the same range as code for other popular processors. Patterson et.al. investigated the use of this encoding in connection with the RISC II Instruction-Cache chip [Patt83].

### 3.4.3 Instruction Set and Machine Performance.

Von Neumann computers get high performance either from fast circuit technology or by exploiting fine-grain parallelism. The latter can be achieved in several ways. One is the "special-case" method. Some frequent combinations of primitive operations are detected by the architect and are made into single instructions. Then, the micro-architect tries to implement these instructions in such a way as to exploit the parallelism available among the primitive operations. Another way is the "general-case" method. A data-path with the desired capabilities and cost is conceived first. Next, the architect defines simple instructions that describe the primitive operations available on the data-path. Then the micro-architect undertakes to pipeline these primitive instructions in such a way that they constantly keep all data-path resources busy.

The "special-case" method has the advantage of requiring less instruction-fetches for the same amount of work; it also has the questionable advantage of allowing better exploitation of parallelism since the particular environment of execution is better known. It has the disadvantages of requiring complex control and of only dealing with special cases. The opposite situation holds true for the "general-case" method. It is more flexible to exploit parallelism, wherever it is available, or to expose the machine capabilities and to allow the compiler/optimizer to make full use of them. Controlling the instruction execution is also simpler. Providing reasonable amounts of pipelining is not very hard, even though the previous and subsequent primitive operations are not known (see chapter 4). The "general-case" has the disadvantage of requiring more instruction fetches.

Architectures with complex instruction sets intend to get high performance using the "special-case" method. Reduced instruction set computers follow the "general-case" approach.

The Berkeley RISC experiment has shown that the differences in code size between the two methods need not be large (§ 3.4.2). On the other hand, it has shown that the differences in size and complexity of the control circuitry is large. While the control section covers 50 to 60 % of the chip area in the M68000 or in the Z8000, it only covers 6 to 10 % of the RISC I or II chip area (see § 4.4 and [Fitz81]). We believe that it is better to spend hardware resources in implementing an instruction-cache, than to spend them in implementing complicated control circuitry with a big micro-program ROM. The reason is that an instruction-cache holds the instructions that are *dynamically* most frequently used, while micro-storage holds the *statically* most frequent primitives, or -- even worse -- some rarely used complex constructs. In RISC I & II, the scarce chip transistors were spent to implement a multi-window register file, since that one has even higher priority than an instruction cache (see chapter 6).

We mentioned above that the "special-case" method may have the advantage of allowing better exploitation of parallelism because of the built-in knowledge of the particular execution environment. However, the opposite may also be true. Micro-programmers sometimes find it hard to correctly optimize all the instructions in a complex architecture. For example, the VAX-11/780 has an *index* instruction used for calculating the address of an array element, and simultaneously checking whether the index fits within the array bounds. The same task can be performed with multiple simpler VAX-11/780 instructions in 45 % less time [PaDi80] ! A similar case for the IBM 370 is reported in [PeSh77]. A sequence of *load* instructions is faster than a *load-multiple* instruction when fewer than 4 registers are loaded.

The Berkeley RISC follows the "general-case" method of pipelining simple instructions. Section 3.3 showed how the memory port is always kept busy and how the register-file and the ALU of RISC II are kept busy all the time except for

the cases of dummy additions, when nothing else could practically be done. More hardware resources make possible the exploitation of more parallelism in RISC-style architectures. Such examples were given in 3.3.2 and 3.3.3 for the case of separate instruction and data memory-ports. One can also consider the possibility of simultaneously dispatching multiple simple instructions when multiple functional units exist. Figure 3.3.3(c) offered one such example. A proposal for parallel dispatching and execution of unconditional-branch and of other CPU instructions will be presented in § 6.3.6.

Comparative measurements of RISC II speed, relative to that of other microprocessors and mini-computers, have shown RISC's superior performance [PaPi82] (also in [PaSe82]). For some of the processors, including RISC, these were collected using a simulator. The average performance ratio from those studies is given below, after being adjusted for the cycle times of the actual RISC II chips (§ 5.2):

Machine:	Basic Clock	Reg-to-reg add	$\left( \frac{\text{Execution Time}}{8\text{MHz RISC II Exec. Time}} \right)$ averaged over 11 programs
RISC II	T=330ns (12MHz)	330ns	0.67
RISC II	T=500ns (8MHz)	500ns	1.00
VAX-11/780	5MHz	400ns	1.7 ±0.9
PDP-11/70	7.5MHz	500ns	2.1 ±1.2
M68000	10MHz	400ns	2.8 ±1.4
BBN C/70	6.7MHz	?	3.2 ±2.2
Z8002	6MHz	700ns	3.3 ±1.3

Five of the above benchmark programs do *not* have procedure calls, they consist of one single function. The execution-time ratio for these programs was given in figure 3.4.1. They show that the performance advantage of RISC is still present, even when the multiple windows of the register file are not used.

More extensive performance measurements were carried out by Miros [Miro82]. He ran the VAX C compiler on both the RISC simulator and on the VAX-11/780. The compilation of three programs took 26 seconds on a 330ns RISC II (simulated), 38 seconds on a 500ns RISC II (simulated), and 50 seconds on the VAX-11/780. It is worth noting that a register-to-register integer addition takes 330ns or 500ns on RISC II, while the VAX-11/780 data-path can perform such an operation in one 200ns *micro*-instruction, even though the execution of a register-to-register add *instruction* takes 400ns on the VAX-11/780.

#### 3.4.4 Overall Evaluation of RISC I & II.

An overall evaluation of the Berkeley RISC architecture must include the multi-window register scheme, the area and transistor statistics of the VLSI implementation, and the human effort that was required to design, layout, and debug the chips.

The evaluation of the multi-window register file was done by Halbert and Kessler and was reviewed in section 2.2.2 (with some further discussion in section 3.2). Here are two more measurements from [PaSe82] for illustration purposes:

<i>Data-Memory-Traffic due to Call's and Return's:</i>			
		PUZZLE	QUICKSORT
VAX-11/780	words	440 K	700 K
	% of all data-mem-ref.	28%	50%
RISC	words	8 K	4 K
	% of all data-mem-ref.	0.8%	1%

These numbers only concern the data traffic due to calls and returns. Further savings in memory accesses are achieved by the default allocation of locals into registers. Thus, one realizes the dramatic savings in memory traffic that the multi-window register file provides. Section 6.1 compares register files to cache

memories, while section 6.2 examines other possible organizations for them. A study of the trade-off between the size of such a register file and its delays due to capacitive loading can be found in Sherburne's thesis [Sher83].

In the RISC I and II NMOS microprocessor chips, the traditional allocation of scarce silicon resources has been radically altered owing to the reduced instruction set. Control circuitry has been drastically reduced, and the silicon area and transistors saved were used for the large register file. The foregoing evaluation showed how the reduced instruction set leads to high utilization of the data-path hardware by the executing programs. This effect is amplified by the faster basic cycle that a simple data-path achieves, as chapter 4 will show. The multi-window register file further enhances performance. The overall result is what we consider to be the most effective utilization of the scarce VLSI resources for performing general-purpose computations. And, last but not least, the human effort required to design, layout, and debug the processor has been reduced by almost an order of magnitude relative to that required for the design of other microprocessors (sect. 4.5). This reduces costs and allows faster exploitation of new and rapidly changing technologies.

We believe that these points prove the viability of Reduced Instruction Set Computer architectures for general-purpose VLSI processors.

## Chapter 3.

## References.

[Bask78] Baskett F.: "A VLSI Pascal machine", Public Lecture, U.C. Berkeley, Fall 78.

[Camp80] R. Campbell: "Compiling C for the Reduced Instruction Set Computer", Master's report, EECS, U. C. Berkeley 94720, December 1980.

- [Cock83] J. Cocke: Informal discussion on the IBM 801 mini-computer, U.C.Berkeley campus, June 1983.
- [DiML82] D. Ditzel, H. McLellan: "Register Allocation for Free: The C Machine Stack Cache", Proceedings, Symp. on Architectural Support for Progr. Lang. and Oper. Systems, Palo Alto, Ca, March 1982, (ACM: SIGARCH CAN vol. 10 no. 2, SIGPLAN Notices vol. 17 no. 4), pp. 48-56.
- [DiPa80] D. Ditzel, D. Patterson: "Retrospective on High-Level Language Computer Architecture", Proc. of the 7th Annual Symposium on Computer Architecture, ACM SIGARCH 8.3, pp. 97-104, May 1980.
- [Fitz81] D. Fitzpatrick, J. Foderaro, M. Katevenis, H. Landman, D. Patterson, J. Peek, Z. Peshkess, C. Séquin, R. Sherburne, K. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer", VLSI Systems and Computations, Carnegie-Mellon Univ. Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI", VLSI Design, vol. II, no. 4, pp. 14-20, 4th qu. 1981; and in: Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.
- [GaVD81] P. Garrison, K. VanDyke: "Compact RISC", CS292R Final Class Report, Comp. Sci., U.C.Berkeley 94720, December 1981.
- [HaKe80] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames", CS292R Final Class Report, Comp. Sci., U.C.Berkeley 94720, June 1980.
- [Henn82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill: "Hardware/Software Tradeoffs for Increased Performance", Proceedings, Symp. on Architectural Support for Programming Languages and Operating Systems, March 82, ACM SIGARCH-CAN-10.2 SIGPLAN-17.4, pp. 2-11.
- [Henn83] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor", Proceedings, 3rd Caltech Conference on VLSI, Pasadena, CA, March 83, Ed. R.Bryant, Comp. Sci. Press, pp. 33-54.
- [Kate80] M. Katevenis: "A Proposal for the LSI Implementation of the RISC I CPU (using a 3-phase clock)", Internal U.C.Berkeley Working Paper, 23 pages, September 1980.
- [Liou83] D. Lioupis: "The RISC II Computer", Internal U.C.Berkeley Working Paper, 25 pages, June 1983.
- [Miro82] J. Miros: "A C Compiler for RISC I", Master's report, EECS, U. C. Berkeley 94720, August 1982.
- [PaDi80] D. Patterson, D. Ditzel: "The Case for the Reduced Instruction Set Computer", Computer Architecture News, ACM SIGARCH 8.6, Oct. 1980, pp.25-33.



- [PaPi82] D. Patterson, R. Piepho: "RISC Assessment: A High-Level Language Experiment", Proc. of the 9th Annual Symposium on Computer Architecture, ACM SIGARCH 10.3, APR. 1982, pp. 3-8; Also in: "Assessing RISCs in HLL Support", IEEE Micro Magazine, Vol. 2, No. 4, NOV. 1982, pp. 9-19.
- [PaSe81] D. Patterson, C. Séquin: "RISC I: A Reduced Instruction Set VLSI Computer," Proc. of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN 9.3, pp. 443-457, May 1981.
- [PaSe82] D. Patterson, C. Séquin: "A VLSI RISC", IEEE Computer Magazine, vol.15, no.9, Sept. 1982, pp. 8-21.
- [Patt83] D. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, K. VanDyke: "Architecture of a VLSI Instruction Cache", Proc. of the 10th Symposium on Computer Architecture, ACM SIGARCH CAN 11.3, pp. 108-116, June 1983.
- [PeSh77] B. Peuto, L. Shustek: "An Instruction Timing Model of CPU Performance", Proc. of the 4th Symposium on Computer Architecture, ACM, IEEE, March 1977.
- [Sher83] R. Sherburne: "Processor Design Tradeoffs in VLSI", Doctoral Dissertation, EECS, Univ. of Calif., Berkeley 94720, Dec. 1983.
- [Shus78] L. Shustek: "Analysis and Performance of Computer Instruction Sets", Doctoral Dissertation, Stanford University, 1977 or 78.
- [Site79] Sites R. L.: "How to use 1000 registers", Proc. Caltech Conference on VLSI, Jan. 1979, pp. 527-532.
- [TaSe83] Y. Tamir, C. Séquin: "Strategies for Managing the Register File in RISC", IEEE Transactions on Computers, vol. C-32, no. 11, November 1983.

## CHAPTER 4:

# THE RISC II DESIGN AND LAYOUT.

This chapter deals with the micro-architecture of the RISC II chip. After a detailed description of the data-path (§ 4.1), it presents the fundamental timing dependencies and the particular timing scheme chosen (§ 4.2) as well as the organization of control (§ 4.3) and some design metrics (§ 4.4). The estimates and the rationale which guided the major decisions are discussed and compared with the picture that emerged after the circuit was designed and laid-out.

### 4.1                      The RISC II Data-Path, and its Use for Instruction Execution.

This section presents the RISC II data-path and the basic trade-offs which were considered during its design. The general form of the data-path is a direct consequence of the instruction set (sect. 3.1) and of the chosen pipeline scheme presented in section 3.3.

A very compact register cell was essential for the implementation of a large register file. Robert Sherburne designed and laid-out such a compact 2-bus register cell by modifying the classical 6-transistor static cell [SKPS82], [Sher83]. The modification allows dual-port read-accesses with single-bus signal-sensing, but requires both busses for a write operation. Dual read-ports and a single write-port perfectly match the basic RISC instruction pattern of reading two registers  $R_{s1}$  and  $R_{s2}$ , and writing the result into a register  $R_d$ . The cell requires a precharge - read - write cycle, which guided us in the choice of the pipeline scheme (figure 3.3.1(c,d)). This cell is about 2.5 times smaller than the 3-bus RISC I register cell, and this feature constituted the main driving force for the development of RISC II.

An arbitrary-amount bidirectional shifter is included in the data-path, as the instruction set specifies. This was designed and laid-out by the present author. It consists of a cross-bar switch made out of pass-transistors [SKPS82]. A compact and versatile lay-out was achieved by routing one data-bus,  $R$ , in the horizontal direction, while the other one,  $L$ , is diagonal, thus providing connection points both on the side and at the top of the shifter module; the control-bus is vertical. The elementary shifter cell is a bi-directional pass-gate that is used in one direction for a left-shift, and in the other direction for a right-shift. The shifter busses need to be precharged before they are used.

A 32-bit integer add/sub ALU, the Program-Counter circuitry, and a pipeline latch complement the basic data-path.

#### 4.1.1 The RISC II Data-Path.

Figure 4.1.1 presents the RISC II data-path. Its basic parts, namely latches, functional units, and busses are the following:



which specifies the byte-within-the-word alignment.

- *NXTPC*: the Next-Program-Counter register, which holds the address of the instruction being fetched during the current cycle.
- *INC*: an incrementer, which computes *NXTPC*+4 (byte addresses).
- *PC*: the Program-Counter register, which holds the address of the instruction being executed during the current cycle.
- *LSTPC*: the Last-PC register, which holds the address of the instruction last executed - or last attempted to be executed. When an interrupt occurs, *LSTPC* will hold the address of the interrupted (aborted) instruction during the first cycle after the interrupt.
- *IMM*: the Immediate latch to hold the 19 LS-bits of the incoming instruction, which contain its immediate constant (if it has one).
- *DIMM*: the Data.In/Immediate combined latch, preceded by the sign-extender/zero-filler. It holds data coming-in from memory, or immediate operands being forwarded to the data-path.
- *OP*: the 7-bit opcode of the instruction, and the SCC and use-immediate bits of the instruction (bits <31:25>, <24>, and <13> respectively; see fig. 3.1.1).
- *busA*, *busB*: the register-file busses.
- *busD*: the bus used for feeding *AI*, and for feeding *DST* from the right-hand side of the data-path.
- *busR*, *busL*: the shifter's busses, optionally connected by the bi-directional cross-bar shifter. *BusR* is also used for feeding *BI*, while *busL* is also used for introducing *Data.In* and immediate constants into the data-path.
- *busOUT*: the bus used for routing addresses and data to the pads, and from there to memory.
- *busEXT*: The off-chip bi-directional time-multiplexed address/data bus, which connects the CPU to the memory. It is electrically identical with the 32 address/data bonding-pads, and with the 32 wires running in the chip, and feeding *RA*, *RB*, *RD*, *IMM*, *DIMM*, and *OP*.

The next subsection explains how these latches, functional units, and busses are used for executing the instructions.

#### 4.1.2 Paths Followed for Instruction Execution.

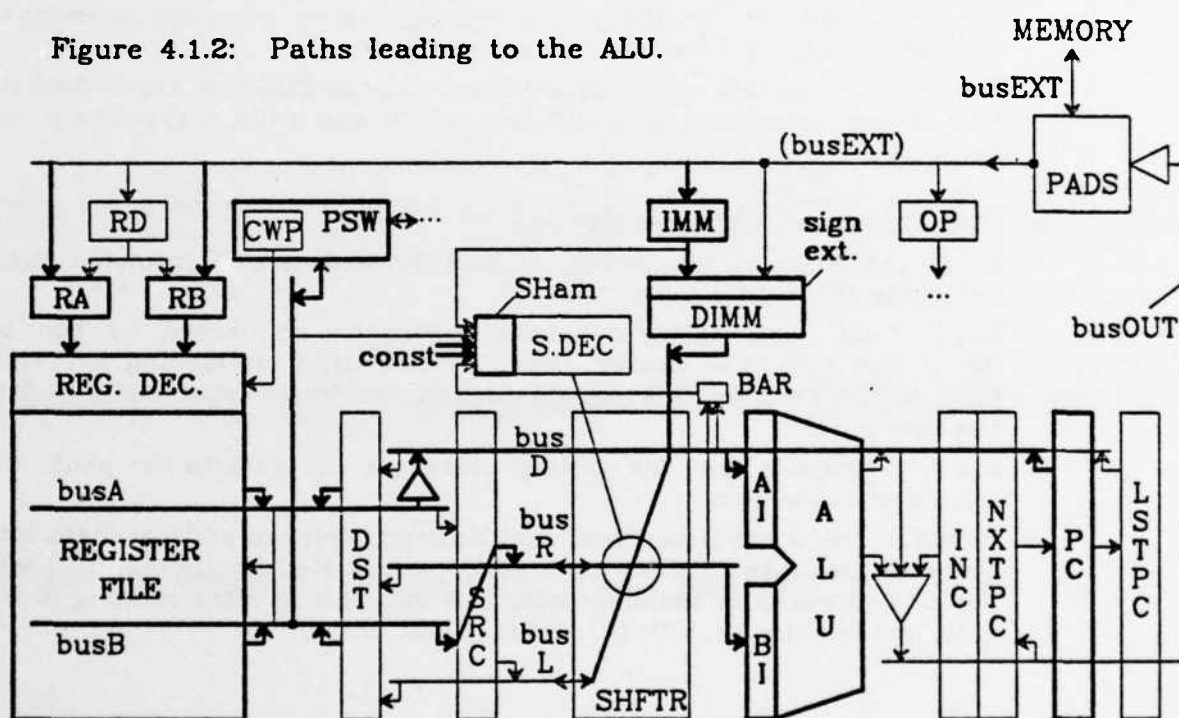
There are few categories of activities that may be going on in the data-path during each cycle:

- The appropriate two sources *S1* and *S2* are routed to the ALU or to the shifter.

- The output of the ALU, of the Shifter, or of the *PC* is routed to *DST*, and is written into its final destination in the next cycle.
- Addresses or data are routed to memory and/or to the *PCs*.

Figure 4.1.2 illustrates how the appropriate two sources *S1* and *S2* are routed to the ALU inputs *Ai* and *Bi*. *S1* may be a register, or it may be *PC* for *PC*-relative addressing. *S2* may be a register or an immediate constant, or it may be *PSW* for the *getpsw* instruction.

Figure 4.1.2: Paths leading to the ALU.



Registers  $R_{s1}$  and  $R_{s2}$  are read through busses *A* and *B*. In case a data-dependency with the previous instruction is detected, internal forwarding occurs (sect. 3.3.1); the *DST* places its contents onto *busA* and/or *busB*. On *getpsw* instruction, reading from the register file is disabled and *busB* is driven from the *PSW*.

The first input of the ALU, *Ai*, is loaded from *busD*. That bus is driven by *PC* or from *busA*, depending on whether a *PC*-relative or a normal instruction is

being executed. The *BI* input of the ALU is loaded from *busR*. That bus is driven from *busB* - through *SRC* - when *S2* is a register or the *PSW*. In that case, the transistors of the shifter are all turned-off, so that *busR* is disconnected from *busL*. When *S2* is an immediate constant, *busR* is fed from *DIMM*, through *busL* and through the shifter. The 19-bit *IMM* latch is connected to *DIMM* in such a way that it feeds the 19 MS-bits of *DIMM*, while the 13 LS-bits of *DIMM* are loaded with zeros. When the instruction contains a 13-bit immediate, the sign-extender converts that into 19 bits. When this MS-aligned 19-bit immediate goes through the shifter, it either stays MS-aligned for *load-high* instructions, or it is right-shifted by 13 and sign-extended (i.e. LS-aligned), for all other instructions.

Figure 4.1.3: Paths leading to the Shifter.

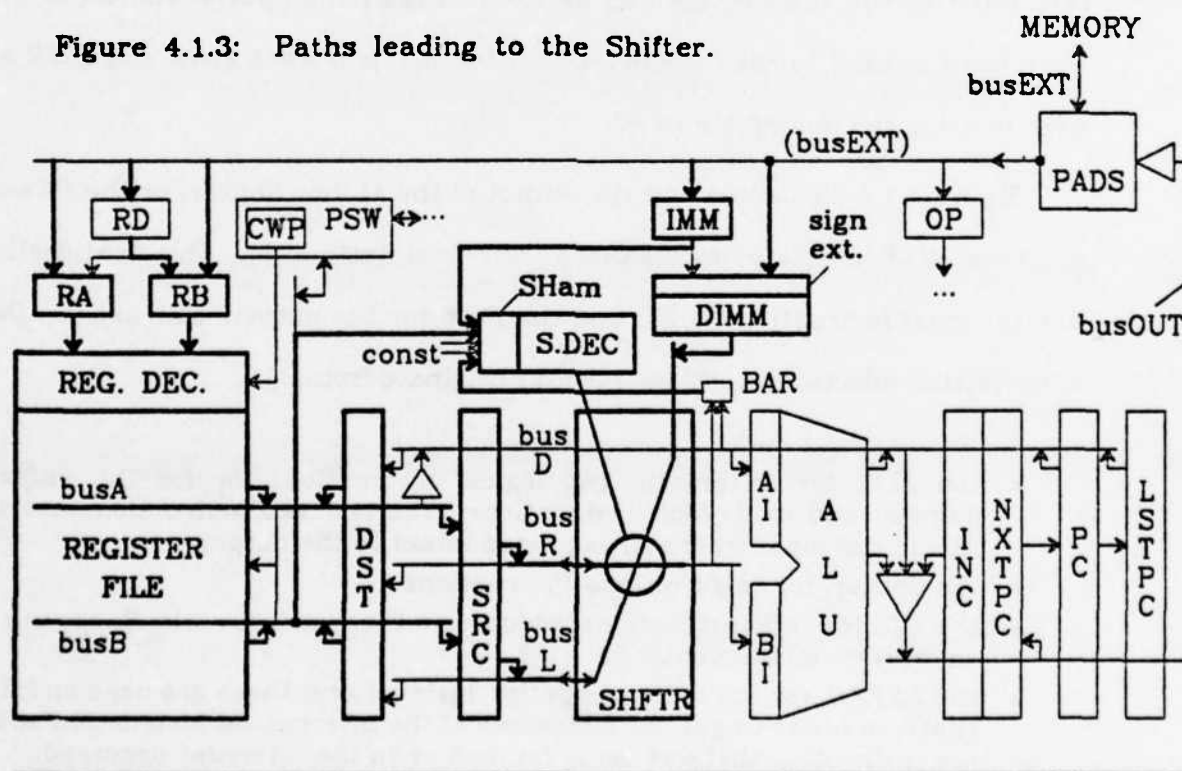


Figure 4.1.3 illustrates how the appropriate sources are routed to the Shifter. For shift instructions, the quantity to be shifted is  $R_{s1}$ , which is read via



*busA* and placed into *SRC*. *SRC* then drives *busR* for right-to-left shifting, or *busL* for left-to-right shifting. The amount of shifting is *S2*, a register or an immediate constant. Thus, *SHam* is loaded with the 5 LS-bits of *IMM* or of *busB* which carries  $R_{s2}$ .

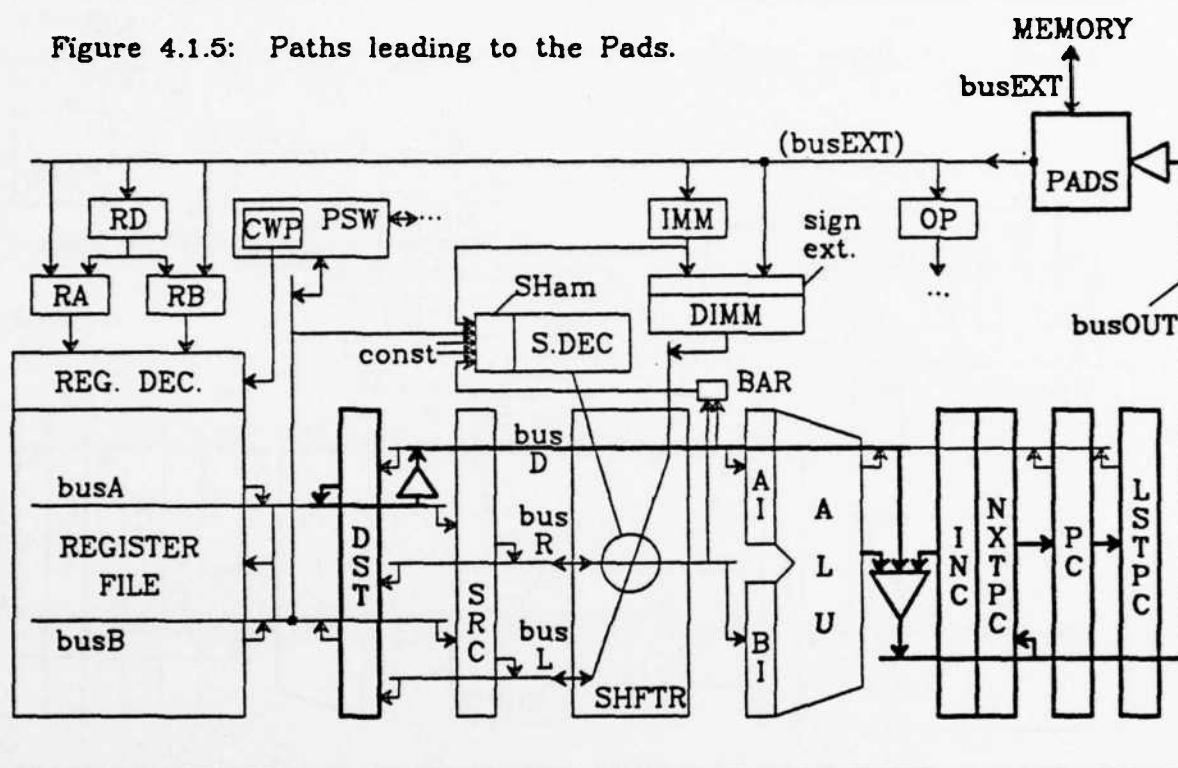
Alternatively, the quantity to be shifted may be data to or from memory, requiring alignment. In that case, the amount of shifting (alignment) is specified by the *BAR*. When data *from* memory must be aligned at the end of a *load* instruction, *DIMM* serves as the shifter's input. Notice that alignment of incoming data requires *left-to-right* shifting only. When data *to* memory must be aligned during a *store* instruction, that data comes from  $R_d$  and is read through *busB* and placed into *SRC*. As it was discussed in section 3.1.2, RISC II limits the addressing modes of *store* instructions to having an immediate *S2*, so that *busB* of the register file can be used to read the data at the same time when *busA* is used to read the index register  $R_{s1}$ , and when *busL* and *busR* are used to bring the immediate to *BI*.

Figure 4.1.4 illustrates how the output of the ALU or Shifter, or the *PCs* are routed to *DST*, and later written into their final destination. This final destination for most instructions is  $R_d$ , and the *PSW* for the *putpsw* instruction. Data to be written into those destinations may originate from:

- the ALU, for arithmetic and logical instructions, or for the *putpsw*, *getpsw*, and *load-high* instructions. The last two instructions use the ALU in the mode where *BI* is passed intact to the output.
- the Shifter, for *shift* or *load* instructions.
- the *PC*, for *call* instructions, which save their address into  $R_d$  for use by the *return* instruction.
- the *LSTPC*, for the *calli* and *getlpc* instructions; these are used on interrupts, in order to get the addresses of the interrupted instruction and of the instruction that was being fetched when the interrupt occurred.



Figure 4.1.5: Paths leading to the Pads.



stages, respectively. During data-memory-access cycles, the whole PC-related circuitry freezes (see sect. 3.3.2 on pipeline suspension).

Addresses for data-memory-accesses always come out of the ALU. Data are sent to memory during *store* instructions. After these data have been read from  $R_d$  and aligned (fig. 4.1.3), they are temporarily kept in *DST* (fig. 4.1.4). Then, *DST* places them on *busA*, which then drives *busD*, and from which they are put onto *busOUT* (figure 4.1.4).

This completes the description of the paths used for the various CPU activities. The complete execution of an instruction is, in general, a combination of some transfer from figure 4.1.2 or 4.1.3, followed by some operation, followed by some transfer from figure 4.1.4 and some from 4.1.5.

### 4.1.3 Trade-offs Considered during the Data-Path Design.

To a large extent the RISC II data-path is a direct consequence of the register-cell used, of the pipeline scheme, and of the instruction set requirements. Some of its important characteristics, however, could be different. Here, we will mention some alternatives, and we will give the reasons for our particular choice. These choices will be evaluated in the next two sections. For a more extensive and detailed study of data-path design trade-offs, and one that particularly addresses electrical design issues, refer to Sherburne's thesis [Sher83].

One trade-off relates to the way immediate constants are brought into the data-path. The shifter is also used for that purpose, in addition to its main function of executing shift instructions and aligning the data on load/store instructions. Shift instructions and data alignment match well with each other, because at most one of these operations occurs in any one cycle, and because both occur near the end of the cycle. However, the routing of the immediates does not match so well with those operations, because it has to occur at the beginning of the cycle, and because it may occur in the same or adjacent cycle with one of the other two functions. In spite of that non-optimal match, a timing solution was found and implemented in RISC II (§ 4.2.2). As a consequence of routing immediates through the shifter, the latter was placed between the ALU and the register file.

There are two possible alternatives to the above scheme. Immediates could be brought into the data-path from the right-hand side, using an extra horizontal bus. This would increase the number of busses crossing the PCs and the ALU, which would cause severe problems for the layout of those densely populated areas. Otherwise, immediates could be brought in with an extra vertical bus just on the left-hand side of the ALU. Aligning the immediate to the LS or MS word-

position could then be done with a 2-way multiplexor at the input of *BI*. This solution is feasible, but it was not chosen because it requires the extra space for a 19-bit vertical bus. The horizontal length of the data-path is severely limited by the desire to have 138 registers, in a chip limited in length by the size of the package cavity.

Another trade-off relates to the way the ALU inputs are fed. In the chosen scheme, multiplexing the ALU sources occurs on the busses which feed *AI* and *BI*. In this way, the ALU inputs are simple latches, and the register-file busses *A* and *B* do not have to extend all the way up to the ALU inputs. This latter fact is advantageous because it reduces bus capacitance thereby speeding-up register-reads. It also alleviates the heavy bus congestion in the *SRC* area. The alternative would be to make the ALU inputs into latches with multiplexors and to extend various busses all the way to them. This scheme would allow registers to be routed directly to the ALU without incurring the extra delay due to the forwarding from one bus onto another. However, it has all the disadvantages corresponding to the advantages of the chosen scheme.

## 4.2

### The RISC II Timing.

This section is concerned with the timing of the RISC II data-path. It starts with a discussion of the fundamental timing dependencies, as implied by the instruction set and the pipeline scheme, regardless of a specific data-path. Then, it proceeds to examine how this timing was cast into specific clock phases for the particular data-path that was chosen (sect. 4.1). Finally, the timing picture that emerged after the data-path was laid-out is presented. Discrepancies

between the three above timing schemes are discussed and explained, and some conclusions are drawn.

#### 4.2.1 Fundamental Timing Dependencies.

Figure 4.2.1 is an abstract timing-dependency graph for the RISC II pipeline (sect. 3.3.). Arrows represent data-path activities, while vertices represent cause-effect dependencies. If an activity *Y* depends on an activity *X* and *must* follow it in time, then the arrow representing *Y* starts from the endpoint of arrow *X*.

The diagram shown in fig. 4.2.1 assumes no knowledge about the data-path, other than the use of a register file with two read-ports, one write-port, and requiring a precharge - read - write cycle. One counter-clockwise revolution around the top half of the diagram represents the main activities occurring inside the CPU during one machine cycle. Equivalently, one clockwise revolution around the bottom half represents the memory cycle occurring in parallel.

Point A illustrates that an ALU or Shift operation can only begin after its source-registers have been read, and that a register-write can only begin after the read operation has been completed. Point B shows that the result of an ALU operation can be used as an effective memory address for a data access or for an instruction fetch. Points C, D, E illustrate a memory read. When this is an instruction fetch, then the path E→G shows that the source-register-number fields of the instruction must be decoded before the corresponding register read accesses can start. The path E→F stands for the alignment and sign-extension/zero-filling needed when bytes and short-words are loaded from an arbitrary memory location into the least-significant position of a register. Thus, point F represents the result of the second-to-last pipe-stage, which is to be written into the destination register during the last pipe-stage (point A). The



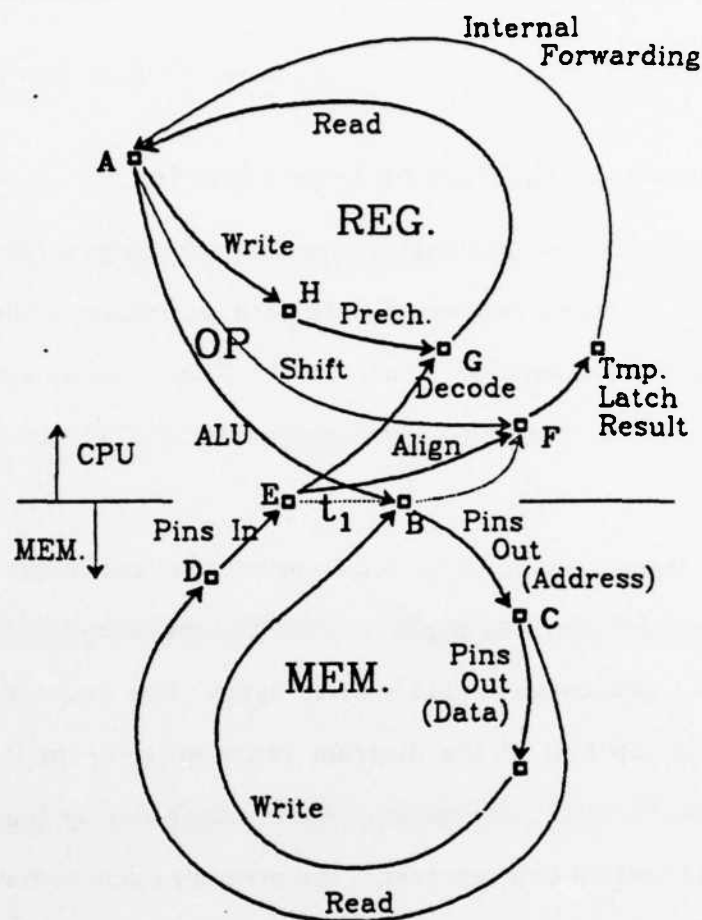


Figure 4.2.1: Fundamental Timing Dependencies in RISC II.

precharge - read - write register-cycle is shown as cycle  $H \rightarrow G \rightarrow A \rightarrow H$ .

Figure 4.2.1 has been drawn with some crude notion of actual time durations in it. The length of the arrows is roughly proportional to the delay of the corresponding data-path activities. Not included is an estimate of the routing delays from one functional unit to another, because no specific data-path is assumed at this point. The diagram shows points E and B separated by an arbitrary amount of time  $t_1$ . Point E represents the end of a memory-read cycle, and point B represents the beginning of the next memory cycle. Because of the



multiplexed address/data pins and because of the non-overlapped memory accesses, point B must be *after* point E, and thus  $t_1$  must be  $>0$ . Within that constraint,  $t_1$  is an arbitrary design parameter that specifies how much the memory access time (B→C→D→E) is shorter than the overall system cycle time.

The diagram shows quite clearly that the internal-forwarding path, F→A, is not critical. The critical paths lie in the register file loop:

$$(precharge)_{HC} \rightarrow (read)_{CA} \rightarrow (write)_{AH}$$

and the figure-8-shaped path:

$$\begin{aligned} & (decode-register)_{EC} \rightarrow (read-register)_{CA} \rightarrow (compute-address)_{AB} \rightarrow \\ & \rightarrow (send-it-off-chip)_{BC} \rightarrow (fetch-instruction)_{CD} \rightarrow (bring-it-on-chip)_{DE} \end{aligned}$$

Thus, using  $T_{cycle}$  for the cycle time, we derive the basic critical path equations:

$$T_{cycle} \geq (reg-prech.)_{HC} + (reg-read)_{CA} + (reg-write)_{AH}$$

$$T_{cycle} \geq (reg-decode)_{EC} + (reg-read)_{CA} + (ALU-add)_{AB} - t_1$$

$$T_{cycle} \geq (pins-out)_{BC} + (mem-read)_{CD} + (pins-in)_{DE} + t_1$$

Thus, the parameter  $t_1$  represents a trade-off between memory and CPU speed. The faster the memory-access time is, the larger  $t_1$  becomes, and the slower the register-decoding and reading and the ALU can be.

#### 4.2.2 The RISC II 4-Phase Timing.

Mapping a timing dependency diagram, like the one of figure 4.2.1, into concrete clock phases for an actual data-path, requires time-area trade-offs to be considered and compromises to be made. For RISC II, a 4-phase clock was chosen, for the following reasons:

- symmetric clock phases are easy to generate;
- register file operation is non-ideal, due to the high resistivity of the polysilicon word lines;
- the register address decoders are simplified;
- the shifter must be used twice per cycle.

Below, we discuss these points and present the utilization of the data-path during the four non-overlapping clock phases. The choices described here were made before the data-path was laid-out and were based on estimates of the various delays. The next subsection, 4.2.3, compares these estimates with the timing picture that emerged after the lay-out was completed and circuits were simulated with their actual parasitic capacitances.

The timing design started with the estimate that a register read takes more time than a register write, which, in turn, takes more time than precharging the register-file busses. Instead of allocating three unequal clock phases for each one of these operations, it was decided to use a 4-phase clock with two long phases  $\varphi_1$  and  $\varphi_3$  ( $\approx 80\text{nsec}$  each), interleaved with two short phases  $\varphi_2$  and  $\varphi_4$  ( $\approx 60\text{nsec}$  each). This would make clock generation easier, since the generator could now have a 2-phase period. Having more clock-phases, of a shorter duration each, is also useful in fine-tuning the timing of the various operations. On the other hand, 4 phases may result in wasted time during the non-overlap periods between clock phases.

To match the register file requirements with the four defined clock phases, register reads were planned to stretch over both  $\varphi_1$  and  $\varphi_2$ , while phases 3 and 4 were allocated to the register-write and precharge operations, respectively. The implications of the high resistivity of the polysilicon word lines was studied carefully. The RC time constant of the address lines can easily reach 50 nsec, causing significant delays between their near and far ends. If a write operation immediately follows a read operation, the read-word-lines may not be fully deactivated by the time writing begins and cause erroneous register file write

operations. To avoid this hazard, it was decided to activate the word-line drivers for read accesses during  $\phi_1$  only. By applying the read pulses to the near end during  $\phi_1$ , these pulses stretch into  $\phi_2$  for the bits at the far end of the word-line.

When a static adder is used in the data-path, it is possible to mitigate the effect of the above RC delay. By placing the least-significant bits of the data-path at the end of the word-lines near the drivers, the adder operation (carry-propagation) can start as soon as these bits are read, without waiting for the most-significant bits from the far end. RISC II, however, uses a dynamic ALU circuit with a precharged carry-chain. That chain can only be released after *all* input bits have settled.

Decoding the register-addresses must be done before the corresponding access begins, so that the word-lines remain stable once activated. The chosen timing scheme has the advantage of allowing the register-file decoders to operate during the phases when the word-line drivers are disabled, that is during  $\phi_2$  and  $\phi_4$ . Thus, no pipeline latches are required at their output, and the congestion in that area is alleviated.

Figure 4.2.2 shows the RISC II timing graph, adapted to the four clock phases and to the data-path of figure 4.1.1. Register reads occur during  $\phi_1$  and part of  $\phi_2$ ; thus, the ALU-operation timing was defined to use the end of  $\phi_2$  for set-up, and  $\phi_3$  and part of  $\phi_4$  for carry propagation. Memory accesses start after the ALU operation has been completed. The effective-address is sent off the chip late in  $\phi_4$  and during the next  $\phi_1$ . Data or instructions come back into the CPU late in  $\phi_3$ , just in time for the instruction and source-register-numbers to be decoded during  $\phi_4$ . For *write* memory-accesses, data are sent to memory during  $\phi_2$ , following the address. This relatively late transfer of the write-data is not a limiting factor for memory chips, because address decoding must occur

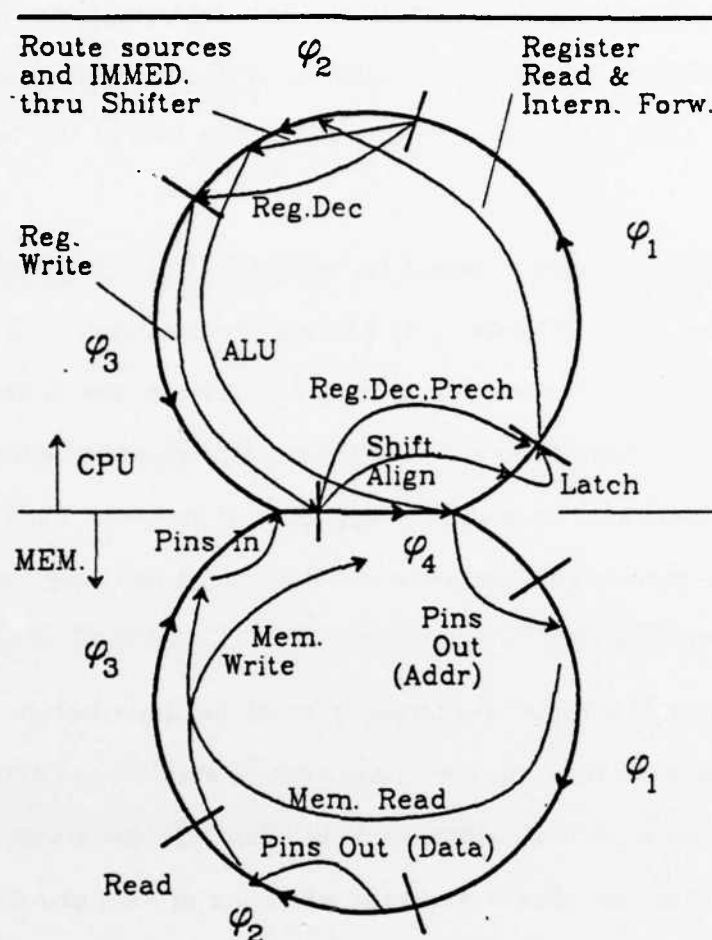


Figure 4.2.2: The RISC II 4-Phase Timing.

before the data are needed.

The choice of using the shifter twice per cycle, for operations with diverse timing (sect. 4.1.3), had important implications. The shifter is used for shifts/alignments and for bringing the immediates into the data-path. The fact that it has to be precharged before each use is another reason why four phases per cycle are necessary.

An immediate constant must be routed through the shifter during  $\phi_2$ . It cannot be routed during  $\phi_1$ , because the previous instruction may have been a *load* that used the shifter during  $\phi_4$  for aligning its data. The routing of the

immediate constant through the shifter may totally overlap the stretched register read operation during  $\phi_2$ . In this case, no extra delays are introduced because of this routing. This was our original estimate and an additional reason why it was decided not to spend silicon area for an extra vertical bus for immediates (sect. 4.1.3). However, this balance of delays of the two operations is strongly dependent on the implementation. The routing of immediates may easily extend beyond the read operation. This would introduce extra delays and routing of the immediates would become the critical path leading to the ALU.

### 4.2.3 The RISC II Timing, Reconsidered after Lay-out.

After laying out the chip, its performance was studied with circuit simulation and analysis, based on the actual sizes and resulting parasitic capacitances of the circuit elements. The critical portions of the data-path were simulated with SPICE2 [NaPe73], using the "worst-case-speed" parameters shown in Table 4.2.1.

Table 4.2.1: SPICE Parameters (worst-case-speed).					
$\lambda$	2.0	$\mu m$	<b>Capacitances:</b>		
	<b>Transistors:</b>		metal	0.14	fF/ $\lambda^2$
$V_{ETo}$	0.9	V	diffusion bulk	0.3	fF/ $\lambda^2$
$V_{DTo}$	-3.2	V	diffusion side-wall	0.3	fF/ $\lambda$
$V_{DD}$	5.0	V	poly over field	0.2	fF/ $\lambda^2$
$V_{BB}$	-2.0	V	gate	1.8	fF/ $\lambda^2$
$\gamma$	0.75	$\gamma^{1/2}$	gate-src overlap	0.5	fF/ $\lambda$
$k'$	20.7	$\mu A/V^2$	<b>Resistances:</b>		
$\mu_0$	800	$cm^2/V \cdot sec$	polysilicon	50	$\Omega/\square$
min. electr. channel L	4.0	$\mu m$	diffusion	10	$\Omega/\square$

The delays through the rest of the chip were estimated using the timing verifier Crystal [Oust83], which utilizes a simple RC model. Figure 4.2.3 shows some results of that study. Two major deviations from the delay values originally expected made the actual timing noticeably different from the ideal picture of

Figure 4.2.1.

The actual length and capacitance of the word-lines is less than originally anticipated. An accurate simulation showed that the "stretching" of the register-read operation into  $\phi_2$  was small. This puts the routing of the immediate-field into the critical path (§ 4.2.2).

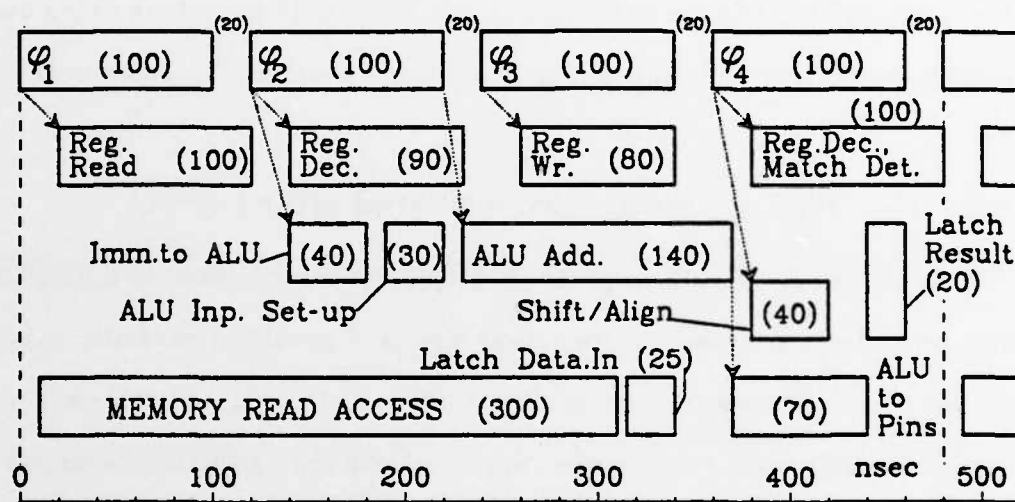


Figure 4.2.3: The RISC II Timing as Simulated after Layout.

Phases 2 and 4 have to be longer than 60 nsec because register-file decoding is slower than expected. RISC II has 138 registers requiring 276 decoding gates in its two-port overlapping-window register file. Minimizing the size and power dissipation of these gates was crucial because of their large number, even though it led to slower operation. This effect compounds to the delay in the decoding circuits due to the overlapping-window scheme (figure 3.2.1(a)). The decoding gates for the non-overlap registers are 6-input NOR gates with a delay of about 30 nsec. (Their low-power pull-up has  $W/L=0.5$ , while it is pulling-up  $\approx 0.25pF$  of load capacitance, consisting of  $45\lambda^2$  of gate capacitance,  $200\lambda^2$  of drain-diffusion capacitance, and a  $160-\lambda$  long polysilicon wire.) However, the

overlap registers require OR-AND-INVERT decoding gates, which have a delay of about 70 nsec. If the circuit of figure 3.2.1(b) had been devised earlier, decoding time for the overlap registers in RISC II could have been reduced to about 40 to 45 nsec.

Another issue studied is the delay resulting from routing data from the register-file across the shifter to the ALU. Driving busses D and R from busses A and B takes approximately 20 nsec. The use of busses D and R to feed the ALU -- instead of extending A and B all the way to the latter -- was chosen because it leads to a much less congested layout in the area of SRC, and because it simplifies ALU input multiplexing (sect. 4.1.3). Extending busses A and B all the way to the ALU would have increased their capacitance and slowed down register-reads by about 15 nsec.

#### 4.2.4 Lessons Learned.

Here we present some insights gained during the design of the data-path. They result from comparing the ideal RISC II timing (§ 4.2.1) with the originally planned real timing (§ 4.2.2) and with the actual timing that finally resulted (§ 4.2.3). It has become clear that loss in performance can be attributed to two main reasons:

- Not enough hardware resources were allocated to frequent operations.
- Too many hardware resources were allocated to infrequent operations.

This is yet another expression of the RISC concept: Capabilities added to a circuit in order to speed up some operation(s) will slow down other operations. Thus, the only capabilities that should be added to a circuit are the ones that speed up the most frequently used operations.



The fine balance of the delays on the word-lines and of routing the immediate through the shifter, that was originally sought, was not achieved. The area saved by passing the immediates through the shifter incurred significant performance penalties. By spending extra hardware, we could have eliminated the need for  $\phi_2$ , and thus significantly speed up the RISC II CPU. The extra hardware is not trivial, but would have been worth spending:

- An extra 19-bit vertical bus would be needed on the left side of the ALU for introducing the immediate constants into the data-path.
- A more complicated register-address decoder would be needed to overlap the write-address decoding with the read operation.
- Pull-down transistors would be needed at the far end of the register word-lines to suppress the read-pulses at the end of the read operation and before the beginning of the write operation.

In general, enough specialized resources should be dedicated to the key CPU operations.

On the other hand, the area occupied by the cross-bar shifter and by its associated input latch/driver, *SRC*, is significant, and so is the bus congestion caused by its busses *R* and *L* in the *DST-SRC* area. This introduces delays into the frequently used path between ALU and the register-file, as discussed at the end of the last subsection. The shifter could have been placed on the right-hand side of the ALU, and immediates introduced into the data-path through a separate bus. It would still consume precious space in the overloaded, critical horizontal length of the data-path, but the data transfer delays would be reduced. In most programs, shifting by an arbitrary amount occurs rarely (chapter 2). Shifting by one or two bits, as required for multiplications and for conversions of array indexes to byte addresses, could be performed in the ALU. Our conclusion is that an arbitrary-amount shifter does not belong in the critical part of the data-path; it could be included somewhere else, accessible only by slower-executing instructions. For example, it could be placed near *DIMM* (fig.

4.1.1) where it would also be useful in aligning data from memory.

## 4.3 The RISC II Control.

The main consequence of a reduced instruction set is the dramatic reduction of the silicon resources required for control. The RISC II opcode-decoder:

- occupies only 0.5 % of the chip area,
- has 0.7 % of the transistors,
- required less than 2 % of our total design and layout time.

This opcode-decoder is the equivalent of the micro-program memory in micro-coded CPU's. The RISC II control section occupies only 10 % of the chip area. These figures stand in sharp contrast to the usual size of control in contemporary microprocessors [Fitz81] [Beye81]:

- the M68000 control section is 88% of the chip area;
- the Z8000 control section is 53% of the chip area;
- the iAPX-432-01 control section is 85% of the chip area;
- the HP "Focus" 32-bit CPU has 78% of its transistors in its microcode ROM.

Although the organization of the RISC II control could be considered as "random logic", it only required half a person-year of design and layout effort.

### 4.3.1 Organization of the RISC II Control.

Figure 4.3.1 shows the organization of the RISC II control. Registers are tagged with a number (1), (2), or (3) to indicate the pipeline stage to which the information they hold belongs. Thus, the registers marked "(1)" hold the

information of the instruction being currently fetched, while those marked "(2)" hold information relating to the currently executing instruction. This latter information will flow into the "(3)-registers" during the next cycle, if necessary. This flow of information among the pipeline registers freezes when the pipeline is suspended for a data-memory-access.

Decoding the incoming instruction is particularly easy in RISC II, because of the simple instruction format with fields of fixed size and position (sect. 3.1.4). RISC II does not have a single physically-integrated instruction register. Instead, it has multiple instruction-field registers, each one close to the place where it is needed.

- Some instruction fields may have multiple interpretations, depending on the instruction. They pose no problem: copies of the same field are latched at *all* places of possible use, and the unnecessary ones are thrown away later on. Examples:
  - Fields <18:14>, <13>, and <4:0> may be parts of an immediate constant, or they may be  $R_{s1}$ , *IMMflag*, and  $R_{s2}$  respectively.
  - Field <22:19> may be part of  $R_d$ , or it may be the jump-condition.
- One instruction field, that must be used as soon as the instruction arrives into the CPU, may originate from two different places in the instruction, depending on the opcode:
  - The second register to be read via *busB* is  $R_{s2}$  (<4:0>) for most instructions, but it is  $R_d$  (<23:19>) for *store* instructions.
 This does pose a problem. The selection of the appropriate origin may not wait until the opcode has been decoded through the normal decoder. Instead, a special fast gate is used to distinguish *store* from *non-store* instructions. The particular choice of opcodes allows this distinction to be made very quickly, based on whether  $\text{instr}<30:29>=11$ .

The register-numbers and the immediate-constant move through the pipeline of instruction-field-registers and are used at the appropriate place and time. Figure 4.3.1 shows their organization and should be compared with figure 4.1.1. The *imm*-field-register-(2) is *DIMM* in fig. 4.1.1. The register-number field-registers-(1) are not shown in fig. 4.1.1, for simplicity. Figure 4.3.1 also shows the circuit which detects data-dependencies and initiates internal-



CPU. These three bits control writing into  $R_4$ ,  $PSW$ , or the  $CCs$ . On interrupts, they get cleared, thus effectively aborting the instruction that was executing †. Also, on interrupts, the op-code latch gets loaded with a special hardwired instruction, *calli*. This instruction calls the interrupt handler, changes the current window (sect. 3.2.2), and saves  $LSTPC$  into  $R_{25}$  of the new (free) window. In this way, the interrupted instruction can be restarted. More information on interrupts can be found in Appendix A.

Besides the 30 decoded-opcode bits, nine more bits of information are involved in the generation of the control signals. They are (see fig. 4.3.1):

- **SCCflag**: set-condition-codes flag from the instruction.
- **IMMflag**: immediate flag from the instruction, specifying whether **short-SOURCE2** is  $R_{s2}$  or an immediate (fig. 3.1.2).
- **Match-Detect**: detection of data-dependencies among second and third pipeline stages, and initiation of internal-forwarding (§ 3.3.1).
- **JUMPcond**: the result of evaluating the condition for a conditional jump.
- **SRCsign**: the sign-bit of the *SRC* latch, used to control sign-extension during *shift* instructions.
- **DIMMsign**: the sign-bit of the *DIMM* latch, used to control sign-extension of immediate operands, or during *load* instructions.
- **BAR**: the Byte-Address-Register, used in detecting address misalignments.

The control signals are generated by ANDing one or more of the above 39 bits with one or more of the clock phases (as in polyphase microcoded implementations). RISC II uses four clock phases, as discussed in sect. 4.2.2. These are externally supplied but the fourth one of them,  $\phi_{4x}$ , is internally "split" into two mutually exclusive phases,  $\phi_4$  and  $\phi_{INT}$ . Normally,  $\phi_4$  is issued; however, when an interrupt occurs,  $\phi_{INT}$  replaces it and sets and clears all the crucial

† An interrupt that occurs during the memory-cycle of a *store* instruction will *not* prevent the memory-write from occurring, if it may occur (i.e. if a page-fault was not the interrupt cause). The same *store* instruction will be re-executed when the interrupt-handler returns, re-writing exactly the same data into exactly the same memory location. Notice that the interrupt cause may *not* have been an address-misalignment, since misalignment-interrupts only occur during the address-calculation cycle.

bits, as discussed above.

There are 100 control signals, half of them for the data-path, and half of them for the control section. These include multiple copies for local uses, clock phases with no control qualification, and decoded-opcode bits with no clock qualification. Most of the 100 timing gates which generate them are very simple:

- 87 timing gates have not more than one clock input:
- 70 timing gates have not more than one clock input and not more than one control-bit input.

Only 18 of the control signals depend on control bits other than the decoded-opcode bits.

Thus, the organization of the RISC II control is simple and straightforward. There is a finite-state-machine with 7 inputs, 30 outputs, and only 2 states. Its outputs are combined with 4 clock phases to generate the control signals. Because not all timing gates are simple 2-input AND gates, and because some control bits are generated outside the FSM, one may find elements of "random logic" organization in the RISC II control. However, that is not the issue. The issue is that that control section has a straightforward organization, that it is easy to understand, and that it required only six person-months of design and layout effort.

### 4.3.3 Simplicity of the RISC II Control.

In RISC II, a few dozen bits of information are enough to control the execution of an instruction. This number stands in sharp contrast to the much larger number of microprogram bits required to execute each instruction in typical microprogrammed machines. We attribute this reduction of required information to the uniformity of the execution of the RISC instructions. All RISC instructions follow the pattern: *read-sources, operate, appropriately-route-the-result,*



and they follow it with the same fixed timing. Thus, the only information which the instruction-decoder needs to generate is whether or not a certain control signal must be activated during the execution of the current instruction. The particular time during which the control signal might be activated is known in advance and hardwired into the gate that drives it.

An important characteristic of the instruction-decoder is its simplicity as a combinational circuit. It decodes 8 inputs which can be in one of 56 relevant states: 23 single-cycle instructions, 16 two-cycle instructions, or illegal (unassigned) op-codes. It has 36 distinct product terms and 30 outputs. The average number of product terms participating in the generation of an output is 1.47. Thus, if it were to be implemented using a general PLA, the OR plane would be very sparse:  $36 \times 30$  crosspoints containing only 44 transistors. For that reason, we used a generalized decoder with a single row of OR gates, instead. This implementation consumes about 60 % less area than the PLA implementation, and it is significantly faster.

We attribute the low number of product terms per decoded-instruction bit to the fact that the instruction set is highly orthogonal, and that orthogonality maps into the op-codes and into the data-path. The low density of assigned opcodes, using only 39 legal opcodes out of 64 (see Appendix A), also had a positive effect. Consider the following examples ( $x$ 's mean "don't care"):



<i>Class of opcodes:</i>	<i>Instructions:</i>	<i>Hardware function controlled:</i>
01xxxxx	load/store	two-cycle instruction: set state-bit to <i>pipeline-suspension-cycle</i>
01xxxx1 0001x01	PC-relative load/store PC-relative jump/call	place <i>PC</i> onto <i>busD</i> on $\phi_2$ ; don NOT place <i>busA</i> onto <i>busD</i> on $\phi_2$
0101x1x	signed load	sign-extender/zero-filler
00011xx	conditional control transf. (jmp/ret)	together with JUMPcond, determines whether to place <i>INC</i> or the ALU output onto <i>busOUT</i> on $\phi_4$

Each one of these classes of instructions is decoded using a single product term. The selection of the 39 opcodes within a  $2^6$  opcode space was done with the purpose of minimizing the size of the decoder; it required only a 2 person-hours effort. The selection of the AND and OR terms of the decoder was done in 2 further person-hours. These numbers demonstrate the simplicity of the RISC II control.

One may consider a circuit with such a small and simple control section as a "hardware engine" rather than a High-Level-Language Computer. However, as section 3.4 has shown, this circuit executes compiled High-Level-Language programs faster than several popular commercial processors. Its compiler and optimizer were easily written, and the machine code is no more than 50 % larger than that of other processors.

## 4.4 Design Metrics of RISC II.

Table 4.4.1 presents several design metrics for RISC II. It gives the total absolute values for the chip area, number of transistors, power consumption, number of rectangles, and approximate design and layout time, as well as the percentages of these values attributed to various CPU sub-functions.

Table 4.4.1: RISC II Design Metrics.								
Part	% Area	% Transistors	% Power (worst case)	Rectangles		Regularity	~% Time	
				% Drawn	% Inst.		Design	Layout
<b>Data-Path: (tot.)</b>	<b>50.</b>	<b>92.6</b>	<b>57.</b>	<b>23.5</b>	<b>90.0</b>	<b>74.</b>	<b>13.</b>	<b>10.</b>
Register File	33.3	73.4	39.3	2.2	70.0	624	3.3	2.3
(storage array)	(27.5)	(64.6)	(34.8)	(0.3)	(60.1)	(3640)		
(decoders)	(5.8)	(6.6)	(4.5)	(1.9)	(9.9)	(104)		
ALU	2.7	5.1	4.0	4.1	4.5	21	2.1	1.1
Shifter	4.3	4.8	2.9	5.3	6.4	24	1.0	1.5
(cross-bar array)	(2.8)	(2.5)	(.0)	(0.1)	(3.4)	(480)	(.5)	(.5)
(inp.latch/dr. decoder)	(1.5)	(2.3)	(2.9)	(5.2)	(3.0)	(11)	(.5)	(1.0)
PC's	2.8	5.6	4.2	3.8	4.7	24	1.3	1.1
Other MUX/latch/drivers	3.2	3.7	6.6	8.1	4.4	11	2.6	3.3
Power wiring	3.9							
<b>Control: (tot.)</b>	<b>10.</b>	<b>5.7</b>	<b>13.</b>	<b>54.4</b>	<b>5.8</b>	<b>2.1</b>	<b>7.</b>	<b>10.</b>
Opcode Decoder	.5	.7	1.0	2.0	.6	7.2	1.1	.6
Instr.&Control Registers	1.6	1.9	4.7	5.5	1.8	5.7	1.5	1.2
CC's, Jmp Cond, Interr.	.8	1.3	2.4	10.5	1.2	2.1	1.3	1.2
Window Number	.5	.5	1.0	4.9	.5	1.8	.5	.4
Timing Gates/Drivers	1.0	1.3	3.9	16.5	1.0	1.1	1.7	1.8
Wiring (non-power)	4.6			13.0	.7	1.0		4.3
Power wiring	.9							
<b>Periphery: (tot.)</b>	<b>40.</b>	<b>1.7</b>	<b>30.</b>	<b>22.1</b>	<b>4.2</b>	<b>3.5</b>	<b>2.4</b>	<b>3.6</b>
Bonding Pads	10.3	1.7	30.0	3.1	2.7	16.6	.8	1.5
Wiring (non-power)	13.0			9.5	.8	1.5	.6	1.7
Power wiring	9.2			1.5	.1	1.0	.6	
Unused area (logo's)	7.2			8.0	.6	1.4		.4
<b>Micro-Archit. Design</b>							<b>9.</b>	
<b>Debugging/Verification</b>							<b>2.</b>	<b>16.</b>
<b>Document. &amp; Overhead</b>							<b>27.</b>	
<b>Total CPU</b>	<b>%</b>	<b>100.0</b>	<b>100.0</b>	<b>100.0</b>	<b>100.0</b>	<b>100.0</b>	<b>60.</b>	<b>40.</b>
<b>Abs. Value</b>	<b>14.9</b>	<b>40.76</b>	<b>1.9</b>	<b>23.5</b>	<b>460.</b>	<b>19.6</b>	<b>5250</b>	
	<b>MA<sup>2</sup></b>	<b>K</b>	<b>Watts</b>	<b>K</b>	<b>K</b>		<b>man-hours</b>	

The register-number decoders and shift-amount decoder were included in the data-path. The control section is subdivided into opcode-decoder, instruction-and-control-registers, and timing-gates (see Figure 4.3.1), as well as other specialized circuits (condition codes, jump condition, interrupt logic, window numbers) and wiring. Areas are separately given for the power wiring (ground and +5V) in the data-path and control sections. For the rest of the metrics, the power wiring of the data-path and of the control is included in their

various sub-blocks, from which it was difficult to separate.

The numbers given for power dissipation use worst-case-power parameters, which are different from those shown in table 3.1:  $V_{TE0}=0.7V$ ,  $V_{DT0}=-3.8V$ ,  $lateral-diffusion = 0.7\mu m$ ,  $k'=30.7\mu A/V^2$ .

The layout tool used was *Caesar* [Oust81], which allows only rectangles with horizontal and vertical edges. We are convinced that the area penalty paid for this restriction was minimal †, and that it was well worth the resulting simplifications in the layout task and in our CAD tools. The number of "drawn rectangles" counts the rectangles explicitly specified in the *Caesar* data base. This number exaggerates the number of rectangles actually placed by the designer. It counts a slightly modified copy of a cell as a totally different cell, as is the case with most timing-gates. The number of "instantiated rectangles" counts all geometry after arrays and calls have been expanded.

Design and layout times are approximate. The totals for data-path and control are higher than the sum of the parts, because they include some general, organizational work. The elapsed time was half a year (times one person) for the micro-architecture design, plus two years (times two persons) for everything else. The total of 5250 man-hours given corresponds to 2.7 man-years, and is lower than the real elapsed time because of other activities occurring in parallel (courses, other research). It does not include work performed after the chip was submitted for fabrication (i.e. more documentation and testing).

In section 4.3 the size of the RISC II control section was compared to that of other micro-processors. Here, we compare the number of transistors, the regularity, and the design and layout effort for the whole chip (data from [Fitz81]):

† The register-cell is limited by fundamental line widths in both directions, and could not be smaller with inclined lines. The shifter's width could be reduced by  $16\lambda$  ( $= 0.3\%$  of the chip width) using  $45^\circ$  lines [SKPS82].

<i>CPU:</i>	<i>Transistors</i> K	<i>Regularity</i> -	<i>Design</i> person-months	<i>Layout</i> person-months
RISC I	44	22	15	12
RISC II	41	20	18	12
M68000	68	12	100	70
Z8000	18	5	60	70
iAPX-432-01	110	8	170	90

When these numbers are combined with the performance comparisons of sect. 3.4.3, the advantages of the Reduced Instruction Set approach become evident.

## Chapter 4.

## References.

- [Fitz81] D. Fitzpatrick, J. Foderaro, M. Katevenis, H. Landman, D. Patterson, J. Peek, Z. Peshkess, C. Séquin, R. Sherburne, K. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon Univ. Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, 4th qu. 1981, pp. 14-20.
- [NaPe73] L. W. Nagel, D. O. Pederson: "Simulation program with integrated circuit emphasis," Proc. 18th Midwest Symp. Circ. Theory, (Waterloo, Canada), Apr. 1973; L. W. Nagel: "SPICE2: A computer program to simulate semiconductor circuits," ERL Memo ERL-M520, Univ. of California, Berkeley, May 1975.
- [Oust81] J. Ousterhout: "Caesar: An Interactive Editor for VLSI Circuits," VLSI Design II(4), Nov. 81, pp. 34-38.
- [Oust83] J. Ousterhout: "Crystal: A Timing Analyzer for NMOS VLSI Circuits," Proceedings of the 3rd Caltech Conference on VLSI, March 1983.
- [Sher83] R. Sherburne: "Processor Design Tradeoffs in VLSI", Doctoral Dissertation, EECS, Univ. of Calif., Berkeley 94720, Dec. 1983.
- [SKPS82] R. Sherburne, M. Katevenis, D. Patterson, C. Séquin: "Datapath Design for RISC", Proceedings, Conf. on Adv. Research in VLSI, M.I.T., Jan. 1982, pp. 53-62.

## CHAPTER 5:

DEBUGGING  
AND TESTING  
RISC II.

This chapter describes the method used and the experience gained in the process of debugging the RISC II logic design and layout and in the functional testing of the RISC II chips. The fact that RISC II chips worked correctly on first silicon is a result of both the simple architecture and the effectiveness of the CAD environment that was used. The fact that RISC II chips were easily tested, without the need to use the scan-in/scan-out loops, is again due to the simple architecture with a readily accessible CPU state.

In this chapter we deal only with logic debugging and functional testing. Timing analysis of the critical data-path and control circuits was done with *SPICE2* [NaPe73]. To check that the timing constraints were met, the timing verifier *Crystal* [Oust83] was used after the whole chip was laid-out. Geometrical layout rules were checked with *Lyra* [ArOu82]. For further discussions on circuit simulation and timing see Sherburne's thesis [Sher83].

## 5.1 Logic Debugging Tools and Methods.

The sophisticated simulation tools available in today's CAD environments make it feasible to debug a VLSI design almost completely *before* fabrication. Such debugging is desirable for several reasons.

Software simulation has a faster turn-around time and lower cost than prototype fabrication, and this is likely to stay true in the next years. Even if this situation should be reversed some day, due to major advances in the implementation of IC chips, software simulation can typically not be avoided. If a prototype returned from fabrication does not perform as expected when plugged into the system or test set-up, how can one find out why it might not work properly? Unless some revolutionary new method of hardware testing is discovered, our capability to monitor or modify the value of internal nodes in a VLSI chip is very limited. Mechanical probing is heavily constrained in terms of number and size of nodes, and because of the capacitive loading introduced into the circuit. Scanning electron microscope methods cannot simultaneously monitor many fast-changing nodes. Software simulation, on the contrary, offers the capability of monitoring and changing the values of internal nodes. This is crucial in complex VLSI systems with limited controllability/observability. Simulation is thus often the only way to gain understanding of the causes of malfunctioning circuits.

Hierarchical design and multi-level simulation tools make it possible to debug a VLSI design at a level of abstraction which is higher than transistors and capacitors. This makes possible a properly structured design employing early debugging, before bad assumptions or errors lead to poor design decisions. For complicated systems, this hierarchical approach also makes the task of debugging manageable by checking each block at the proper level of abstraction.

The RISC II design was done at four levels:

- architecture (ISP),
- micro-architecture (RTL),
- gates (logic), and
- layout (circuit & mask geometry).

The architecture level corresponds to the system specification. The RISC I & II architecture was described using the *ISPS* notation [Corc80] [BaMa78]. The corresponding simulator was used to test the architecture, but it was too slow to run large programs. A "special purpose" RISC simulator program was written [Tami81] and was used in conjunction with the RISC compiler and assembler for evaluating the RISC performance.

The micro-architecture level corresponds to a register-transfer description, roughly like the one given in chapter 4. The *Slang* language and simulator [VDFo82] were used for this level, as described below in § 5.1.1.

At the gate level, we only used diagrams on paper. Because that level is quite close to the final layout, the lack of machine-readable description was not important.

At the layout level, *Esim* [TermES] was used for switch-level simulation of the circuit that was extracted from the layout, as described below in § 5.1.2 and § 5.1.3.

### 5.1.1 SLANG: Simulation and Debugging at the RTL Level.

*Slang* is a LISP-based hardware description language and event-driven-simulator [VDFo82] that is suitable for describing and simulating digital systems at mixed levels of abstraction. We did use it in such a mixed-mode description and simulation:



- *Gate-level*: Some parts, such as the timing-gates in the control section, were described at the gate-level.
- *Gate-Vector-level*: Some latches in the data-path, whose proper internal operation needed verification, were described as two cross-coupled 32-bit vectors. Bitwise boolean operations on 32-bit integers were used for that purpose.
- *Register-level*: Other latches were described at the register-level, by using assignment operations on 32-bit integers.
- *Block-level*: Parts of the system were described as a block, using a LISP program. Such was the case for the off-chip memory, the register-file and shifter plus their decoders, the opcode decoder, the interrupt logic, and the jump-condition PLA.
- *Real-Polarity-level*: The data-path busses were described using their real polarities. This was done in order to verify the correctness of the design, since some of them are used with different polarities at different times.
- *Symbolic-Polarity-level*: For other values, no actual polarity was specified; they were just simulated as *ON* or *OFF*.
- *Symbolic-Value-level*: Some of the node-values were symbolic constants or lists of objects. Such was the case for opcodes and instructions, which were both described at the assembly level. This mnemonic representation was very helpful. It was easy to implement because of the LISP environment.

Difficulties were encountered in describing and simulating bi-directional pass-transistors, like the ones in the register-cell and in the shifter. To solve this problem, *Slang* was modified to permit multiple sources driving the same node, provided that one of them is characterized as "stronger" than the others. Then, the bi-directional transistors were modeled as two simultaneous connections in opposite directions, with each one transmitting a value in its own direction. This has the undesirable side-effect of creating a feedback loop in the simulated circuit, which is not present in the real circuit. In our case, the storage effect that the feedback loop gives did not bother us, because the corresponding real busses do exhibit dynamic capacitive storage, and because external sources, which are "stronger" than the pass-transistors, can break the loop. However, this fictitious loop effect may be a real problem when this same simulation technique is applied to other systems.

*Slang* allows nodes to have an "unknown" value and uses this value during initialization and when conflicts of multiple sources arise. The technique was

very useful in making sure that the RJSC II chip can be initialized using the external pins alone. A minor problem had to be overcome in a few cases. When only some of the bits of a word are unknown, *Slang* will assign the value *unknown* to the whole word. These words thus had to be split into multiple parts, and the corresponding hardware into multiple nodes, leading to a more cumbersome description.

*Slang* should normally be used during the micro-architecture design, before layout work begins. In our case, however, it was used only after the data-path was laid-out. No errors in the data-path design were found, owing probably to the processor's simplicity. On the other hand, 5 to 10 design errors were uncovered in the control section and were corrected before layout began. Simulation for debugging was done using multiple small test-programs as input. The correctness of the design was checked by manually looking at the simulation results. This manual checking was not very time-consuming, and thus no need was felt to use the architecture simulator for automatic result checking. The set of test programs was believed to be fairly complete. Nevertheless, later during the switch-level simulation a design error was found that was not covered by the test-programs (an instruction setting the carry-bit, immediately followed by an instruction using the carry-bit). This shows that our ad-hoc approach to test generation is not satisfactory.

### 5.1.2 Node Naming and Circuit Extraction.

Throughout the layout, a conscious effort was made to flag as many of the nodes in every cell as possible with names. This practice turned out to be very useful for documentation and for debugging. About half a dozen layout errors were uncovered early in the debugging phase merely through the analysis of the node-naming error diagnostics of the circuit extraction program. The preferred

names were the ones used for the corresponding nodes in the *Slang* description. Good and consistent naming conventions proved to be important but were not easy to devise in the early stages of the design. Polarity information was included in as many of the names as possible.

We used *Mextra* [FitzME] to extract the transistor and interconnection list from the layout. The program issues two types of diagnostic messages:

- (1) reports of the discovery of two different names on the same electric node, and
- (2) reports of the same name appearing on more than one node.

Some of those messages correspond to legal situations. Situation (1) may arise when a node has multiple functions, and (2) may result from cells that are replicated many times for different bits or functional units. Other messages correspond to layout errors. Type-(1) messages indicate accidental short-circuits or erroneous wiring of output *A* to input *B* and of output *B* to input *A*. Type-(2) messages indicate missing connections if there are more node instances with a certain name than there should normally be according to the design. *Mextra* follows a certain naming convention that allows it to separate type-(1) messages into legal and erroneous cases, and to report them separately. However, no similar mechanism is available for messages of the latter type.

### 5.1.3 Co-Simulation at the RTL and Extracted-Switch Levels.

The transistor and interconnection list that was extracted from the layout, was simulated with the switch-level simulator *Esim* [TermES]. A number of unconventional circuits had to be looked at carefully, to decide whether their simulation would present any problems. If necessary, the circuit-description file was hand-patched to overcome such problems:

- The shifter presented no problem, because *Esim* knows how to handle bi-directional transistors.
- The register cell itself, presented no problem, but its interface to the busses did. Firstly *Esim* would not detect the cell-disturbance that could occur if a read-operation were attempted via a non-precharged bus; the reason is that *Esim* believes that a static pull-up is always stronger than a capacitive load. Secondly, *Esim* could not handle internal forwarding correctly, because in the real chip that depends on the *DST* driver being stronger than the register cell; *Esim*, on the other hand, always assumes that a pull-down is stronger than a pull-up.
- The bootstrap-drivers for the register word-lines could be simulated correctly, with the only exception that an *unknown* decoder output would cause the *unknown* value on its word-line to propagate onto the bootstrapping clock. The reason is again the lack of understanding that a strong pull-up is present.
- Some static latches in the control section have a long depletion transistor which is used as a feedback resistor. *Esim* considers all depletion transistors with gate and source tied together as pull-ups (!), and thus fails to simulate those latches correctly.

A simulator that understands that some transistors are stronger than others would solve the above problems, provided that it also handles depletion transistors correctly.

Debugging a system at the switch level is very difficult, because of the large number of nodes that have to be watched, and because the correct values that these nodes should have are not always obvious. For that reason, *Slang* has been written in such a way that it can execute together with *Esim*. Two lists of "corresponding nodes" are defined by the designer. *Slang* will drive the *Esim* nodes on the first list with the values that their corresponding nodes have in *Slang*, then perform a simulation step at both levels, and then compare the values that the "corresponding nodes" of the second list have in *Slang* and in *Esim*, and print any discrepancies. In this way, the results of the switch-level simulation are automatically checked against the debugged RTL description. During RISC II debugging, the values of 1300 circuit nodes (single bits) were being compared for equality after each clock phase transition. Besides the ease provided by the automatic checking, the method is also very helpful in identifying the cause of a discrepancy and finding the offending layout error. When

checking is done automatically, the values of many nodes throughout the chip can be checked, and thus, the first discrepancy reported is usually very close to its cause.

The switch-level simulation uncovered 11 mistakes:

- one timing design error on a scan-in/scan-out loop (this was not uncovered by *Slang*, because *Slang* does not know about these loops);
- one missing flip-flop design error (this is the one mentioned at the end of § 5.1.1, that the *Slang* test programs weren't testing for);
- one flip-flop not being cleared on  $\phi_I$ , as it should;
- an error in the programming of a decoder;
- one connection to the wrong point;
- one case of reversed connections; and
- five cases of connections to the wrong polarity.

This shows that most of the errors were cases of wrong connection. *Mextra's* name-checking did not catch them either because the node naming was incomplete, or because *Mextra* only looks at the names on electricly connected points and does not check the consistency of input and output signal names on simple gates.

## 5.2 Testing the RISC II Chips.

The RISC II layout was submitted for fabrication to MOSIS at  $\lambda=2\mu m$ , and to XEROX PARC at  $\lambda=1.5\mu m$ . Twenty-eight chips were received back from MOSIS two months later, and five chips were received from XEROX in one and a half months. Five out of the 28 MOSIS chips were rejected by visual inspection, and 3 of the remaining 5 that were bonded and tested were found functionally correct. The fastest one of them run at a 500 nsec cycle-time. One out of the 5 XEROX

chips was found functionally correct, except for some bad bits in a few registers. It run at a 330 nsec cycle-time.

All the digital IC's designed in our group at U.C.Berkeley during the last three years have been debugged by simulation of the extracted layout. Our experience has invariably been that chips carefully debugged in this way are functionally correct on first silicon. This was true for all of the following big projects:

- RISC I,
- FFT cordic rotator (Lioupis, Wold),
- RISC II,
- RISC Instruction Cache

It demonstrates the viability and the effectiveness of this debugging method.

The present section deals with the functional testing of the RISC II CPU chips. It describes the hardware set-up and the testing strategy that was used, and it discusses the usefulness of scan-in/scan-out loops.

### 5.2.1 Testing Set-up and Strategy.

Figure 5.2.1 shows the set-up that was used for testing the RISC II CPU chips. A Digital Analyzer (Tek DAS-9100) was used for pattern generation (PG), for data acquisition, and for comparing the acquired to the expected values. We preferred the use of an external clock generator, rather than synthesizing the clock phases with the pattern generator. It reduces the number of different patterns that need to be generated, since the chip itself only needs a new pattern every 4 phases. The external clock generator also allows short non-overlap periods and individually-variable clock phases. External tri-state buffers were required, because the particular DAS that was used offered no tri-state PG channels.



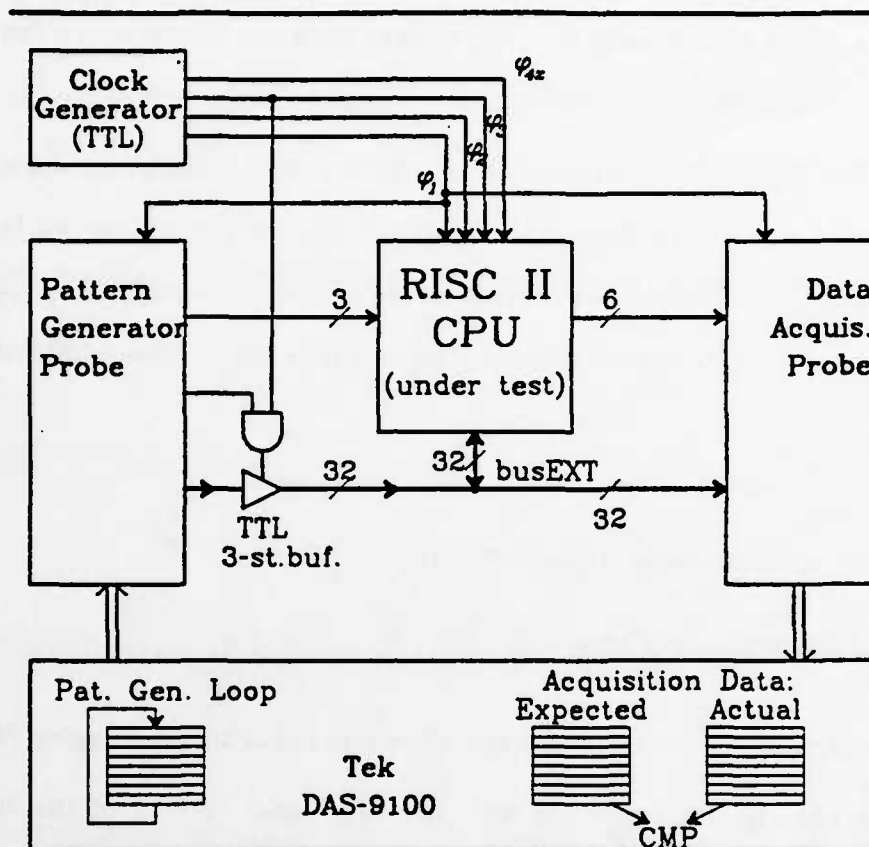


Figure 5.2.1: RISC II Testing Set-Up.

Testing is done by having the chip execute normal instructions, which are supplied by the PG. Data for *load* instructions are also supplied by the PG. The address stream produced by the CPU chip is recorded, and compared to the expected one. Data originating from *store* instructions can also be recorded, by using  $\phi_2$  as the acquisition clock.

Because of the simple architecture, the small CPU state, and the single-cycle execution of instructions, the RISC II CPU chip has very good controllability and observability. Regardless of its previous state, the chip can be initialized to a useful state in only 3 cycles:



- hold the *RESET* pin high for one cycle ( $NXTPC \leftarrow 80000000_H$ );
- *putpsw*:  $PSW \leftarrow R_0 + imm$  (initialize *CWP*, *SWP*, interrupts);
- *add*:  $R_d \leftarrow R_0 + imm$  (initialize register  $R_d$ ).

Of course, it would take about 150 cycles to initialize all 138 registers, but one only needs to do that when testing for defects in the register-file. Once a few registers have been loaded, instructions can be tested, and the result of any instruction can be read from the pins in one cycle:

- *jump-indexed* always to  $R_d + R_0$ .

where  $R_d$  is the destination register of the instruction to be tested. (*Getpsw*, *getlpc*, and PC-relative instructions can be used to read the *PSW* or the *PC*'s). Except for the *PSW*, the *PC*'s, and the registers, all other storage devices in the CPU are initialized and used within the execution cycle of any instruction that uses them. Thus, they are directly controllable and observable by that instruction.

The test programs used were similar or identical to those used during debugging with *Slang* (§ 5.1.1). The same comments apply here. The tests performed are believed to be fairly complete, but there is no proof of that. Our ad-hoc approach to the problem of test generation is simply due to the lack of a good theory and suitable CAD tools.

### 5.2.2 Scan-In/Scan-Out Loops.

Scan-in/scan-out (SISO) capability [WiAn73] [EiWi77] [FrSp81] is added to chips in order to increase the controllability and observability of their internal state from the pins. It can be implemented by organizing all latches as shift-registers and by providing serial ports for reading and writing into them. In the RISC II CPU, three latches, situated at central positions, have SISO capability: *DST*, *SRC*, and the latch holding the output of the opcode-decoder. The two

former ones form a single 64-bit loop, while the latter one forms a 32-bit loop by itself. Each of these two loops uses separate dedicated pads and wires for their shift-clock and serial-I/O. In this way, if the normal connections of the CPU with the external world are defective, the loops may still provide access to the chip's interior.

The cost of the SISO loops in the RISC II CPU is not high: Area-wise, they consume only about 1% of the data-path, about 3% of the control section, and 10% of the pads. Speed-wise, they slow the machine cycle down by about 1%. However, the cost of using the loops for testing the chip is high. First, to load values into them, or to read their contents, 32 or 64 cycles are required, as opposed to the 1 to 4 cycles required for accessing the same latches with normal instructions (§ 5.2.1). Second, reading/writing via the loops requires that the normal CPU clocks be stopped and that the SISO-shift clocks be activated. Third, the whole mode of operation of the SISO loops is much different from the rest of the chip, thus requiring significant human effort and additional software tools for their use.

There are several situations in which chips can be put in a test set-up:

- (1) for debugging a design;
- (2) for debugging a fabrication process, and finding out what particular circuit is not working in a series of defective dies; or
- (3) for identifying operational dies for packaging and use.

In our view, number (1) above should be used only to *verify* that the design is correct; debugging should be done in software, as discussed earlier in this chapter. Number (2) above can be done with dies specifically designed for that purpose; it does not need to be done with production chips. Thus, we believe that the usefulness of SISO loops should only be considered in the context of product testing, i.e. purpose (3) above. SISO loops can always increase the

controlability and observability (C/O) of those dies where a certain type of defects would have disconnected the SISO latches from the external world without the presence of the serial path. However, for correctly designed and defect-less dies, an increase in testability is not always present. In our view, SISO loops should be evaluated according to how much they increase the C/O of *operational* chips.

In RISC II, the data-path SISO loop offers no increase in C/O in a correctly working chip, because normal instructions can also be used to copy values between registers and *DST* or *SRC*. The control SISO does increase the C/O of the opcode-decoder's output, because normal instructions cannot read that output; neither are there any instructions that can load an arbitrary value into these latches. However, we consider that increase in C/O to be of limited usefulness. The bits in that latch control so many things throughout the CPU, that even if only one of them is incorrect, chances are that most instructions will not work at all. From that point of view, the observability of the latch is very high. On the other hand, loading an arbitrary pattern into that latch -- one that does not correspond to any real instruction -- is of very limited usefulness. It is normally very difficult to devise new ways of getting a data-path to perform useful transfers that do not already exist; this is especially true in RISC II, where the timing information is hardwired in the timing gates.

For all the above reasons, the SISO loops in the RISC II CPU have *not* been used in testing it, just like it had happened with RISC I [FoVP82]. For these reasons, we consider this style of design-for-testability to have limited usefulness in the case of micro-architectures with readily accessible internal state. For bus-oriented chips, with reduced C/O of their internal state, we suggest that an alternative style of design-for-testability be considered before resorting to SISO loops. When a latch is close to some bus, consider connecting it to that bus for

test purposes; this may not be more expensive than adding SISO capability. Such a parallel connection will usually be faster and easier to use than a SISO loop. Of course, if all latches are connected to a single bus, they will all become inaccessible if that bus does not work; but - again - we are only interested in testing for working chips. For chips that are dominated by random logic and that have few or no busses, SISO loops may still be a good solution.

## Chapter 5.

## References.

- [ArOu82] M. Arnold, J. Ousterhout: "Lyra: A New Approach to Geometric Layout Rule Checking", 19th Design Automation Conf. Proceedings, ACM-IEEE, June 1982.
- [BaMa78] Barbacci, Mario, et.al.: "The ISPS Computer Description Language", Carnegie-Mellon University, 1978.
- [Corc80] G. Corcoran: "Using ISPS to Specify the RISC I Architecture and Implementation", Master's report, EECS, U. C. Berkeley 94720, December 1980.
- [EiWi77] E. Eichelberger, T. Williams: "A logic design structure for LSI testability", Proceedings, 14th Design Automation Conference, pp.462-468, ACM, New York, June 1977; also in J. Des. Autom. Fault-Tolerant Comp., 2.2, pp.165-178, May 1978.
- [FitzME] D. Fitzpatrick: *Mextra*: a Manhattan circuit extraction program, U. C. Berkeley. See e.g. "1983 VLSI Tools - Selected works by the original artists", report No. UCB/CSD-83/115, Comp. Sci. Div., U. C. Berkeley, CA 94720, March 1983.
- [FoVP82] J. Foderaro, K. VanDyke, D. Patterson: "Running RISCs", VLSI Design, vol. III, no. 5, pp. 27-32, Sep/Oct. 1982.
- [FrSp81] E. Frank, R. Sproull: "Testing and Debugging Custom Integrated Circuits", ACM Computing Surveys, Vol.13 #4, pp.425-451, December 1981.
- [NaPe73] L. W. Nagel, D. O. Pederson: "Simulation program with integrated circuit emphasis," Proc. 16th Midwest Symp. Circ. Theory, (Waterloo, Canada), Apr. 1973; L. W. Nagel: "SPICE2: A computer program to simulate semiconductor circuits," ERL Memo ERL-M520, Univ. of California,

Berkeley, May 1975.

- [Oust83] J. Ousterhout: "Crystal: A Timing Analyzer for NMOS VLSI Circuits,"  
Proceedings of the 3rd Caltech Conference on VLSI, March 1983.
- [Sher83] R. Sherburne: "Processor Design Tradeoffs in VLSI", Doctoral Dissertation, EECS, Univ. of Calif., Berkeley 94720, Dec. 1983.
- [Tami81] Y. Tamir: "Simulation and Performance Evaluation of the RISC Architecture", Electronics Research Lab. Memo. UCB/ERL M81/17, Univ. of California, Berkeley, CA 94720, March 1981.
- [TermES] C. Terman: *Esim*: a switch-level simulation program, Massachusetts Institute of Technology.
- [VDFo82] K. VanDyke, J. Foderaro: "SLANG: A Logic Simulation Language",  
Research Project (Master's Report), CS, U.C. Berkeley, June 1982.
- [WiAn73] M. Williams, J. Angell: "Enhancing testability of LSI circuits via test points and additional logic", IEEE Trans. on Computers, C-22.1, pp.46-60, January 1973.

## CHAPTER 6:

# ADDITIONAL HARDWARE SUPPORT FOR GENERAL-PURPOSE COMPUTATIONS.

Chapter 2 studied the nature of general-purpose computations as expressed in von Neumann languages. It was seen that a few simple operations account for most of the execution time and that high performance depends mostly on

- exploitation of fine-grain parallelism,
- fast addressing and operand accessing,
- fast decision making and branching, and
- fast floating-point operations (in numeric applications).

This suggests that it is more effective to use special devices that provide fast access to instructions and operands than to use precious chip area for the implementation of complex instructions.

The Berkeley Reduced-Instruction-Set-Computer experiment, which was presented in chapters 3, 4, and 5, has investigated this direction in computer architecture. RISC I and II provide pipelined execution of simple instructions in an environment where local variables are readily accessible. Section 3.4 gave an evaluation of the experiment, showing both the viability and the advantages of

simple instruction sets.

Any hardware resources that remain available after a pipelined data-path and its simple controller have been implemented should be spent as effectively as possible for increasing performance. According to chapter 2, this means providing fast access to the most frequently used operands, fast compare-and-branch operations, and fast number crunching for numeric applications. The first two of these issues are considered in this chapter. The latter one -- number-crunching -- is not considered in this dissertation.

Enhancements intended to providing the above support should be included in a processor according to a "priority list" that depends on the hardware resources (e.g. silicon area) available at a given time. We believe that these priorities are as follows:

1. Register File for frequently used scalar variables,
2. Instruction Cache with support for fast decision making,
3. Data Cache for non-scalar operands.

This ordering results from their relative cost and pay-off. A register file, even of modest size, for scalar variables allows high performance gains. An instruction cache is larger in size but is essential in feeding a fast data-path with new instructions. A data cache has less of a visible effect in an architecture where many of the operands, namely the scalar variables, are already in registers. Separate instruction and data caches are proposed here for two reasons. First, independent memory ports are desirable for parallel instruction-fetching and data-accessing (§ 3.3.2, 3.3.3); second, each one of these two cache types can be structured in a different way to take best advantage of the peculiarities of its usage. Such organizations are proposed below in § 6.3 and § 6.4, after the issue of fast access to scalar variables has been discussed in § 6.1 and § 6.2. Section 6.5 deals with another important issue: moving data into and out of a processor's



main memory.

## 6.1 Multi-Window Register Files versus Cache Memories for Scalar Variables.

Section 3.2 presented the organization of the RISC multi-window register file and explained how it provides fast access to the frequently used local scalar variables. That register file acts as a small and fast buffer for the top of the stack of procedure activation records, that is, for the most recently used local variables. In that respect, a multi-window register file is similar to a cache memory. This section investigates the similarities and the differences between them.

### 6.1.1 The Various Kinds of Locality of Reference.

The locality of the memory references made by a program has usually been studied by statistical methods in a "black box" approach, without looking at the underlying program properties which cause it (see for example [Smit82]). However, a study of the way programs access memory, like the study in chapter 2, shows that this locality has interesting properties arising from the nature of computations. Memory references can be distinguished into three categories, with different locality properties each:

- *Instructions:* Instruction-fetches are read-only accesses. They are sequential in small blocks -- between *if* or *call* or *loop* statements. Locality arises from the repeated accesses to instructions inside loops. Since programs spend most of their time in small inner loops, this locality is

high.

- **Scalar Variables:** Scalars occupy a single memory location, and hence, that fixed location is accessed whenever the variable is used. As noted throughout chapter 2, some scalar variables are heavily used during execution. These tend to be few in number, declared locally in their procedure, and used as array-indexes, counters, pointers, flags, or temporary storage locations. For example, in the critical loop of *fgrep* (fig. 2.4.1), out of the 29 operand accesses per iteration, 18 are made to 4 local scalars, 2 are made to 1 global scalar, and 9 are made to non-scalars. These facts show the high locality of the references to the few scalars in critical loops. They are not unrelated to the way the human mind works by hierarchically breaking big tasks into smaller ones, and by only dealing with a few objects/concepts at each level.
- **Non-Scalar Variables:** Arrays and structures occupy many memory locations each, and accesses are *not* made to the same location each time. Usually, certain elements of a few non-scalar variables are accessed once or a few times, and then accesses shift to "neighboring" elements of those variables (chapter 2). "Neighborhood" here, may or may not be actual proximity in virtual address space, depending on the type of the accessed data structure.

Cache memories usually treat all memory references alike and base their operation on the average statistical observations. Some computers have dedicated cache(s) for instructions and/or data. The rest of this chapter (except for § 6.5) deals with alternatives to this organization.

### 6.1.2 Comparison of Registers and Caches for Scalars.

The organization of the circular buffer of  $N$  register windows in the RISC architecture (§ 3.2) is such that the  $N-1$  most recent procedure activation records are kept in that register file (except for the parts of activation records which do not fit in one register window). A cache memory of size  $M$  blocks, on the other hand, is organized so that the  $M$  most recently used memory blocks are kept in it. Since, in those organizations, procedure activation records are kept in a LIFO memory stack, it follows that those parts of the most recent activation records that have actually been used in the recent past are kept in the cache memory. Thus, a multi-window register file approximately holds a subset of what a cache memory holds, and both of those devices are fast

memories intended to provide quick access to their contents. The similarities and the differences among the two approaches will be further investigated here.

A register file will hold *all* the local scalars of a procedure, or a random subset of them in the rare case where there are more than can fit into a window. A cache, on the other hand, will only hold those local scalars which have actually been used in the recent past. In this respect, the cache memory is better, since it adapts itself *dynamically* rather than statically to the demands of the computation. Two negative effects, however, make this adaptability worse than what it might be theoretically. First, whole *blocks* containing the most frequently used scalars are kept in the cache, not just the words themselves. This increases the probability that the precious on-chip memory locations hold unused data. Second, most caches are set-associative with a small set-size. In that case, other data (or instructions) may overwrite some of the recently used local scalars. The difference in the adaptability of register files and caches is also reduced by the fact that most procedures have a few local scalars and use them heavily (§ 2.2.2), so that the static prediction is not far from the dynamic situation. On the other hand, fixed-size window schemes waste part of the window when the activation record is small (see next section).

At this point, it is worth discussing the issue of global scalars, as well. There are usually many global scalars, but only a few of them are heavily used. For example, *fgrep* (§ 2.4.1) has 20 global scalars declared, but only one of them is used in the critical loop. Cache memories will dynamically discover those variables and hold them. For a compiler, on the other hand, it is very difficult -- if not impossible -- to determine which ones are frequently used and to allocate them in registers. A viable approach requires the programmer to give hints to the compiler, using *register-type* declarations like the ones that C allows for local scalars. The same declarations would be useful for those few

procedures which have more local scalars than a window can hold.

While the above comparisons did not show a decisive difference between a cache and a register file, these differ strongly when *addressing overhead* is considered. Caches offer addressing transparency at the machine-language level, at the expense of always referencing objects by their full, long identifier (address). Registers offer no such transparency at that level, but they allow referencing of objects by short identifiers. Addressing transparency for registers is offered in the HLL domain, which is all that matters for the programmer. The effect of the identifier size is very important, both in terms of accessing delay, and in terms of hardware requirements.

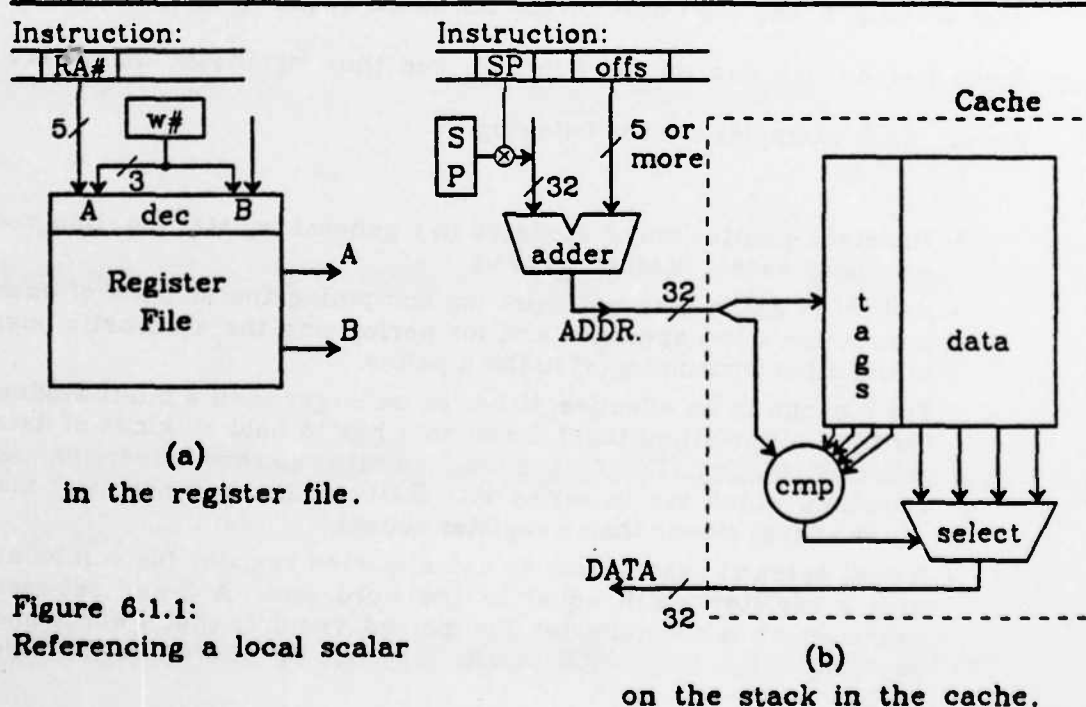


Figure 6.1.1:  
Referencing a local scalar

Figure 6.1.1 illustrates these points. To reference a local scalar in the RISC II multi-window register file, 8 bits of information are decoded in a specialized decoder, and one out of 138 registers is activated and places its contents onto

the corresponding bus. To reference a local scalar on the execution stack, the stack-pointer *SP* must be selected and gated into an adder, where the short offset constant out of the instruction must be added to it. This is a *long addition* which produces a full-length memory address. The cache uses the LS-part of that address to access a wide RAM, in order to read a number of words and tags equal to the set size. The MS-part of the address is compared to the tags, and one of the words that were read is selected.

It is clear that the long address addition required to access a local scalar on the stack makes such cache references slower than a register reference, even if the cache itself is equally fast as the register file. Pipelining can alleviate that, but the gains are limited by the occurrence of jump instructions. In most practical situations, the high cost of the hardware required to make cache references fast enough can not be afforded, and thus significant delays are introduced. Such examples are the following:

- The stack-pointer may be located in a general register file, thus requiring a register access for it to be read.
- Dedicated ALU's may not exist for computing the address of each of an instruction's two operands and for performing the arithmetic operations of the other instruction(s) in the pipeline.
- For a cache to be effective, it has to be larger than a multi-window register file (not counting tags), because it has to hold all kinds of data -- not just local scalars. This fact, combined with the tag comparison and word-selection, which are in series with RAM-reading, will normally make the cache access slower than a register access.
- A dual operand read-access to a dual-ported register file can be achieved with a register width equal to the word size. A 2-way set-associative cache, which is normally not dual-ported, requires that 4 words and 4 tags be read. Such large RAM widths may not be affordable for an on-chip cache.
- The communications bandwidth requirements are higher for the cache, due to the full-width memory address. This is an important bottleneck for off-chip caches.

The combined effects of the above points can be seen in the following table. It compares the speeds at which the PDP-11/70 and the VAX-11/780 can access

local integers in their registers and in their cache:

	PDP-11/70	VAX-11/780
cache access time (hit)	300ns	200ns
register access time ( $\mu$ -instr.)	?	200ns
$i:=j$ ; $i, j$ : reg.	300ns	400ns
$i:=j$ ; $i$ : reg., $j$ : stack (hit).	1050ns	800ns
$i:=j$ ; $i$ : stack (hit/write-thru), $j$ : reg.	1500ns	1200ns
$i:=j$ ; $i, j$ : stack (hit/write-thru).	2250ns	1400ns

The superiority of multi-window register files over cache memories, for keeping scalar variables, is thus clear. There are two fundamental reasons for that. Firstly, scalars differ from data-structures in both their properties and their usage. Scalars are few, they are referenced repeatedly, and they are used to *name* data-structure elements (e.g. pointers, array-indexes). Secondly, when the sources of computations can conveniently be segregated into different storage devices, parallel access to them becomes possible. Such is the case with instructions, scalars, and data-structures.

## 6.2 Fixed-Size, Variable-Size, and Dribble-Back Multi-Window Register Files.

The RISC I and II register file organization has fixed-size windows, and copies these windows completely to/from memory when overflows or underflows occur (§ 3.2). That is not the only possible organization for a multi-window register file. Two alternative schemes are discussed in this section: register files with variable-size windows and "dribble-back" register files that save and restore windows "in the background" in parallel with normal instruction execution.

### 6.2.1 Variable-Size-Window Register Files.

Allocating procedure arguments and local scalars into registers is possible, in the RISC window scheme, because the number of those scalars is quite small most of the time, and they can thus all fit into the current window. The measurements in [HaKe80] (§ 2.2.2) showed that the number of these arguments and locals is smaller than 13 in more than 95% of the executed procedure calls. The RISC II register file has enough space for 15 arguments and local scalars in each of its fixed-size windows (the 16th register is used for the return-PC).

While few of the procedures would need more registers than a window has, many of the procedures use only a few of the available registers in a window. According to the dynamic measurements in [HaKe80], a procedure activation record needs  $4.6 \pm 1.3$  registers on the average for its arguments and locals. According to similar - but static - measurements in [DiML82], that number is 5.7 words per procedure on the average †. What this means is that a large portion of a fixed-size window remains unexploited most of the time, if that window is large enough for most of the activation records to fit in it. According to the above numbers, two thirds of the RISC II registers remain unutilized, on the average. Thus, sizable silicon resources are wasted; this is a serious drawback of the fixed-size window scheme.

---

† Both measurements include all locals — scalars and non-scalars. However, the averages given here only take into consideration those procedure activations which required  $\leq 24$  words for their arguments and locals, because it may safely be assumed that larger requirements arise only out of local non-scalars. The first number is the average of the 9 dynamic averages for the 9 measured programs. The second number is the average over the 1400 statically defined procedures in all of the standard UNIX commands (/usr/src/cmd/\*.c).



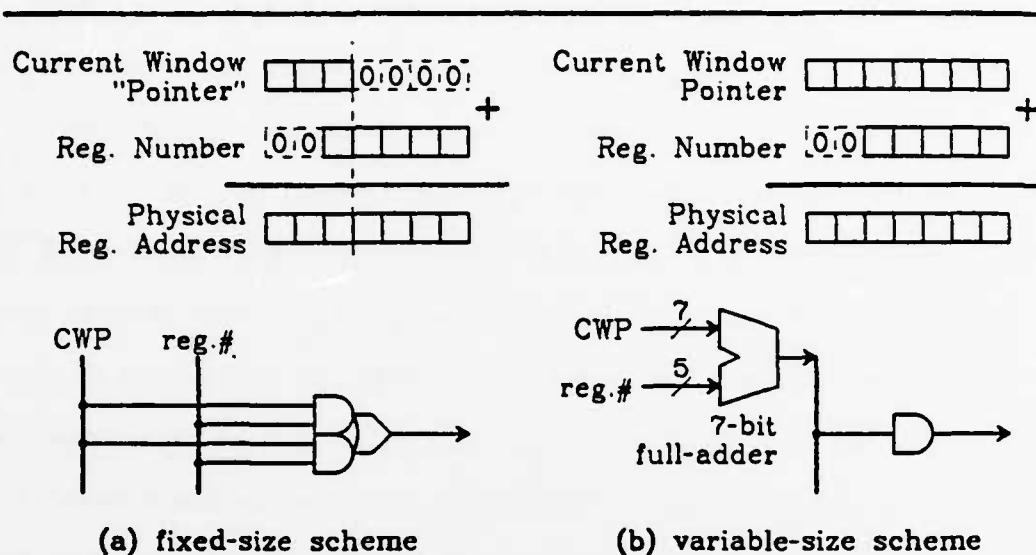


Figure 6.2.1: Register Decoders for Multi-Window Schemes (concept and implementation).

The alternative is to use a register file with *variable-size* windows in which each window is only as large as is needed. Overall, such a register file needs fewer registers, because of the improved utilization. This has several desirable effects:

- transistors are freed and can be used for other functions on the chip,
- the register file is faster, due to its smaller size and correspondingly smaller parasitics (see [Sher83]), and
- saving and restoring registers into/from memory is faster, since no unused registers are copied.

Of course, a maximum window size is always imposed by the total register-file size and by the number of bits available in the instruction format for specifying a register-number. However, in this variable-window-size scheme, whenever a child procedure is called, the current window pointer only moves from its previous position by as many registers as the parent procedure actually uses, instead of moving by a fixed predefined distance. In this scheme, which is closer to the

traditional stack of activation records, windows may "begin" (be aligned) on any arbitrary register in the register file. That means that the Current Window Pointer (*CWP*) must now have single-register resolution in pointing to the beginning of the current window. The register addressing process can no longer be done with a simple AND-OR decoder — an addition must be performed, instead. Figure 6.2.1 illustrates these points. The required 6- or 7-bit adder in series with the register decoder will slow-down the decoding of register-numbers, which is on part of the critical path of the execution phase (fig. 4.2.1). Nevertheless, the delay of a carefully designed small adder needs not be much longer than the extra delay caused by the OR-AND-INVERT gates required for decoding the overlap registers in the fixed-size scheme (§ 4.2.3). This, coupled with the smaller and faster register file, may make the variable-size scheme quite attractive.

Another penalty that must be paid in the variable-size window scheme is the additional overhead per call-return pair for updating the *CWP* and checking for overflows or underflows. These tasks cannot be carried out by hardwired decrement/increment and compare operations as in RISC II. Instead, the number to be subtracted or added to *CWP*, and the distance of *CWP* from *SWP* to be checked for over/under-flow detection, depend on the number of arguments and locals of the parent or child procedures. These pairs of procedures may be separately compiled, in languages like C, and thus their respective requirements are not known at compilation time. Either the linkage editor should be used to patch the call/return statements, or an additional instruction per call-return pair is required. Figure 6.2.2 shows two of the available options for updating and checking *CWP* in the variable-size scheme.

- In (a), *CWP* points to the base (the "beginning") of the current window. When a parent calls a child, *CWP* is changed by the size of the parent's frame, and availability is checked for a maximum-size new window. These can both be performed by the *call* instruction. When the child returns,

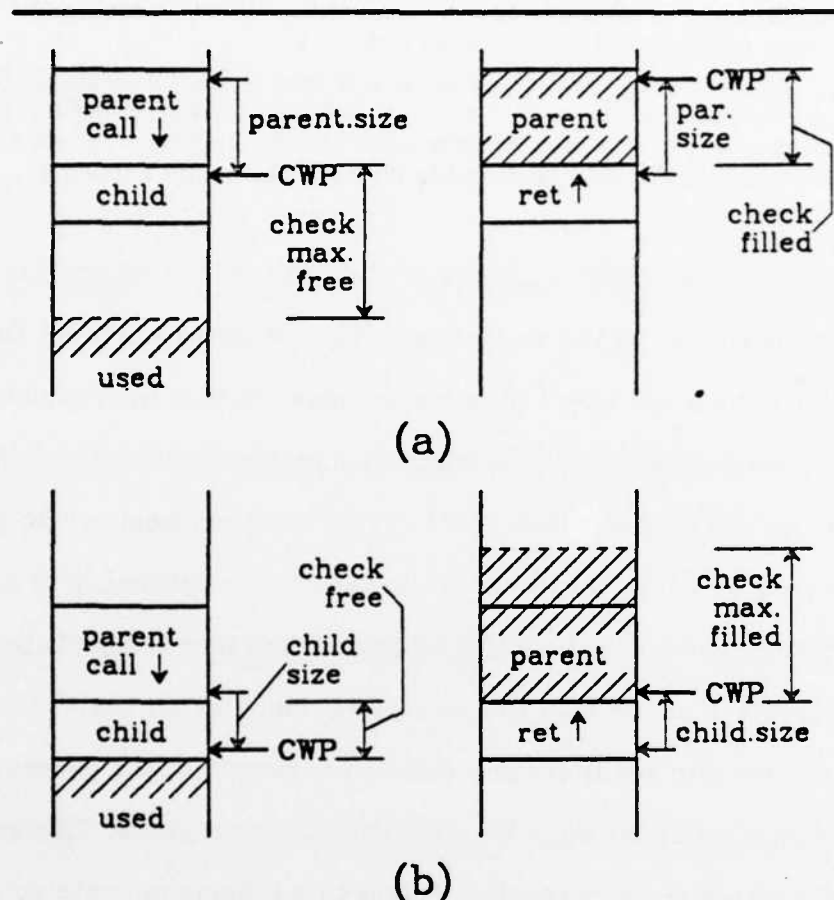


Figure 6.2.2: Update-&-Check Options  
with Variable-Size Scheme.

*CWP* has to be changed by the size of the *parent*'s frame again. However, that size is not known to the child procedure which executes the *return* instruction, unless the linkage editor patches the code. In the absence of such patching, an extra instruction has to be inserted after each *call*, to "catch" the return, to update *CWP*, and to check whether the parent's registers are still present in the register file. Alternatively, the parent's *CWP* value may be passed to the child along with the return-PC.

- Part (b) of figure 6.2.2 shows the second option, in which the *CWP* points to the "limit" of the current window. In that case, *CWP* has to be changed by the size of the child's frame upon calls and returns. When the *return* instruction is executed, that size is known, and no problem exists (the check for validity of the parent's registers is made for the maximum possible size of the parent's frame). However, the *call* instruction is executed by the parent, and thus an extra instruction has to be inserted at the entrance of the child procedure. This scheme inserts statically less extra instructions (dynamically both schemes execute the same number of extra instructions), but when the child checks for free space it must check for the sum of its own frame plus the maximum number of outgoing

arguments of all of its *call*'s. With this scheme, passing the parent's *CWP* along with the return-*PC* does not work.

- A third option, used in Ditzel's C Machine Register-Stack [DiML82], is to insert extra instructions both at the entry-point of every procedure and at the target of every *return* instruction. In this way, an accurate check for over/under-flow is possible on both *call*'s and *return*'s.

Ditzel's "Stack Cache Register Set" for the C Machine, described in [DiML82], is similar to the variable-size window scheme, except that the *CWP* is extended to be a full 32-bit memory address. In that way, registers are always accessed with their equivalent memory address. To avoid the high penalty of performing the long addition  $CWP + offset$  once per register access, that addition is performed at the time the instruction is fetched into the instruction cache. This scheme exploits the statistical fact that many of the non-recursive procedures are called with the same *CWP* many or all the times. When this is not the case (with recursive procedures for example), the procedure code is re-fetched into the cache upon the new procedure activation. The accessing of the registers with memory addresses makes this scheme be quite similar to a cache memory (§ 6.1.2). Its fundamental difference from a cache is that it is only used for the top of the execution stack. In this way, parallel access to it and to the rest of the memory is possible, and it is managed as a single circular buffer with no address tags, no LRU replacement, and no set-associativity.

### 6.2.2 Dribble-Back Register Files.

In the multi-window schemes that were examined up to now, saving and restoring windows to/from memory was only done on overflows and underflows respectively. These schemes are successful when enough windows exist, so that overflows and underflows rarely occur. An alternative scheme is to perform the saving and restoring *before* a need for it arises and to perform it "*in the background*", that is, in parallel with normal instruction execution. As long as this

background copying does not slow down program execution, it doesn't matter how frequently windows are saved or restored. Thus, it is possible to have very few windows in a register file that is managed with this method. This kind of management was proposed by Sites in [Site79], who used the name "dribble-back" to describe it.

The advantage of a dribble-back register file is that it can be small in size, and thus fast in operation ([Sher83]). Also, in the ideal case, it will never overflow or underflow. Its disadvantage is the high memory bandwidth which it requires for saving/restoring registers in parallel with normal instruction execution. Thus, dribble-back register files are attractive for high-performance systems, where the cost of the extra bandwidth may be affordable. To provide sufficient bandwidth, one might use a pipelined cache that permits two accesses per machine cycle -- one for the executing instruction and one for the saving/restoring process. Alternatively, separate instruction and data caches may be utilized. In processors employing register windows, the data-memory-port is often left idle. In the measurements reported in § 3.2.2, only about one fifth of all executed RISC instructions were *load* or *store*. The idle memory cycles can be used in the background for saving or restoring registers.

Figure 8.2.3 shows a dribble-back register file with two windows -- the minimum possible size.

(a) shows a data-movement organization selected for ease of understanding the operation of the scheme. Upon execution of a *call* instruction, the local and input-argument registers of the parent procedure are copied (saved) into a back-up set of registers (window). Simultaneously, the output-argument registers are copied into the input-argument ones. Now, the child procedure can start executing, with its input arguments in the input-registers, and with the local and output registers free to be used. In parallel with the child's execution, the back-up window is copied (saved) to memory, in preparation for the event of another *call* occurrence. If that other *call* does not occur, then upon execution of the *return* instruction by the child, the input-registers are copied into the output-registers, thus returning values to the parent, and the back-up set of registers is copied

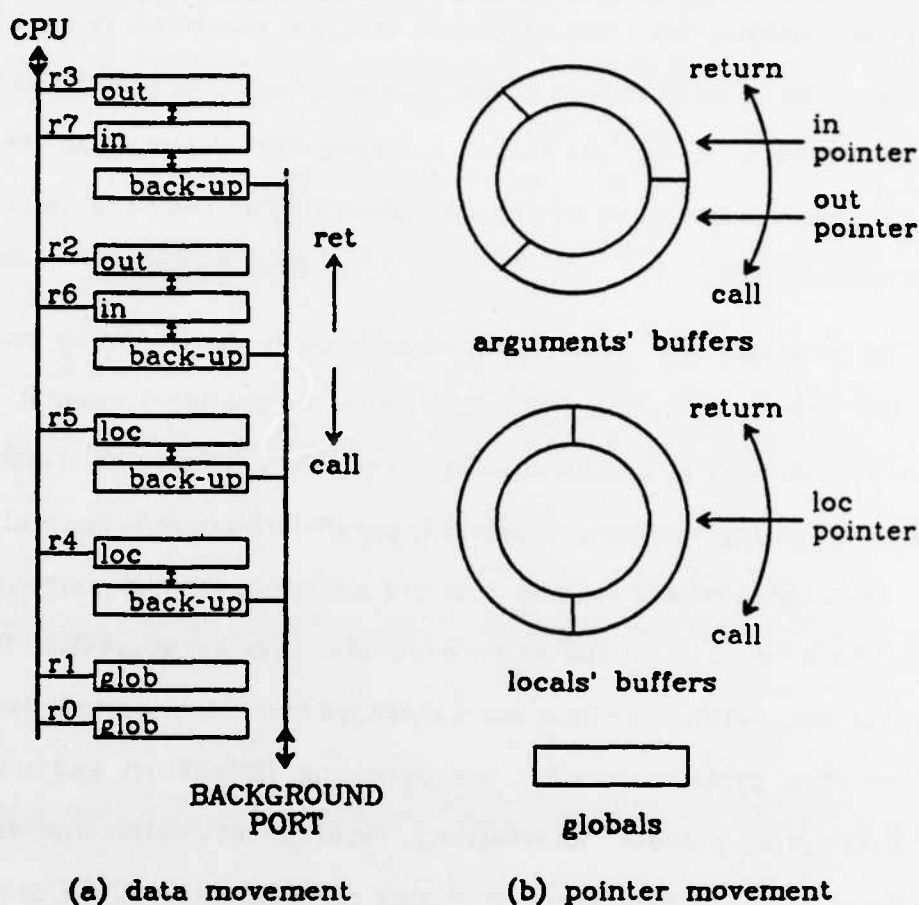


Figure 6.2.3: Dribble-Back Register File.

into the local and input registers, thus restoring the parent's activation record. Now, in parallel with the parent procedure continuing execution, the back-up window must be prepared for the event of another *return* instruction; the locals and input-arguments of the grand-parent must be copied (restored) from memory into those back-up registers.

Thus, the following management schedule is followed: After a *call* instruction, prepare for a new *call* by saving the parent's frame. After a *return* instruction, prepare for a new *return* by restoring the parent's frame.

Part (b) of figure 6.2.3 shows an organization that is preferable for implementation. Here, pointers are changed, instead of moving data from one window into another. In this organization each register cell only needs connections to



the CPU bus(es) and to the background-port bus. In organization (a), register cells need connection to one less bus, but they need additional *shift-type* connections to their neighbours. These shift-type connections are more expensive than normal bus connections in terms of silicon area; thus register-file (b) is more compact than file (a). Another disadvantage of organization (a) is that all registers are copied on *call/return*'s, thus requiring extra power for data transfer. For this reason, organization (b) was also preferred for the RISC register file.

The main advantage of dribble-back register files is their smaller physical size, resulting from the low number of windows required. Successful performance of the minimum-size dribble-back register file is critically dependent on whether enough time is usually available between two successive procedure calls for a window to be saved in memory, and between two successive procedure returns for a window to be restored from memory. To evaluate this, the profiled code of the *sed* and *mextra* programs (see § 2.4.2 and § 2.4.4) was analyzed by hand, yielding the following dynamic measurements:

- When procedures are called and start executing:
  - $\approx 1/2$  of them call no further children;
  - $\approx 1/3$  of them call another procedure after executing 0 to 4 HLL statements (that is 0 to 10 machine instructions); and
  - the remaining  $\approx 1/8$  of them call another procedure after executing 8 to 10 HLL statements.
- After a child returns to its parent:
  - in 50 or 70 % of the cases, another procedure is called in a while;
  - in 35 or 10 % of the cases, the parent returns after executing 0 to 1 HLL statements ( $\approx 0$  to 3 machine instructions); and
  - in 15 or 20 % of the cases, the parent returns after executing 4 or more HLL statements.

These numbers mean that, with a 2-window dribble-back register file, roughly 30 % of the calls or returns will have to wait because the back-up window is not yet ready — unless the background-coping memory port has a bandwidth of several



words per machine cycle. Thus, effective multi-window register files with very few windows are not easily achieved with the dribble-back scheme.

### 6.3 Support for Fast Instruction Fetching and Sequencing.

The process of fetching instructions performs two basic functions:

- supplying "fuel" -- i.e. instructions -- to the execution unit, in order for the computation to proceed, and
- guiding the computation onto the proper path, according to decisions dynamically made in the execution unit.

In a high-performance processor, where simple instructions control a pipelined data-path, it is important for both of these functions to be fast. This section investigates hardware and architectural support for achieving that goal.

The various organizations proposed in this section are centered around the use of an instruction cache. As mentioned in the introduction of this chapter, an instruction cache is one of the desirable hardware enhancements for a high-performance processor, for several reasons:

- a cache for instructions is an effective device, because it exploits the locality of references arising out of loops in programs;
- a cache that is dedicated to instructions is simpler than a general cache, because it is read-only;
- an instruction cache which is separate from the data-memory port of the CPU is desirable for allowing parallel instruction and data accesses (§ 3.3.2);
- an on-chip instruction cache utilizes the silicon area more effectively than microcode ROM, because it dynamically adapts its contents to the requirements of the executing program.

- an independent instruction cache lends itself to incorporation into an instruction fetch-and-sequence unit (see below).

An alternative to an instruction cache is a single or multiple instruction buffer. Such buffers are simpler than a general cache and rely on the usual small size of critical loops. However, the effectiveness of buffer schemes is limited by the fact that each iteration of a critical loop often consists of the execution of several small *non-contiguous* blocks of instructions, rather than of a single contiguous block that could fit in an instruction buffer. As an example, the small critical loop of *fgrep* (§ 2.4.1), which consists of the execution of only 11 lines of source code, actually extends over two pages of source program. Since about one out of 4 to 6 executed instructions is a successful conditional branch or call/return, the average size of blocks of contiguously executed instructions is only about 4 to 6 instructions (see § 6.3.2, § 2.2.1).

### 6.3.1 Remote-PC Instruction Units.

The instruction fetching process enjoys a significant degree of independence from the computation process. That independence is the basis of a desirable hardware partitioning into separate fetching and execution units, allowing for both locality of information processing and parallelism of operation.

Figure 6.3.1 shows an organization that minimizes the communication bandwidth between an instruction fetch-&-sequence unit and the unit which executes instructions. The Program Counter (*PC*) is contained in the former unit, hence the name "*remote-PC*" scheme. The fetch-&-sequence unit understands and executes control-transfer instructions -- jumps, calls, and returns. For that unit, two characteristics of the instruction sequencing mode are important:

- *Conditional/Unconditional Sequencing:* For conditional transfer instructions, a 1-bit condition -- supplied by the execution-unit -- selects one of

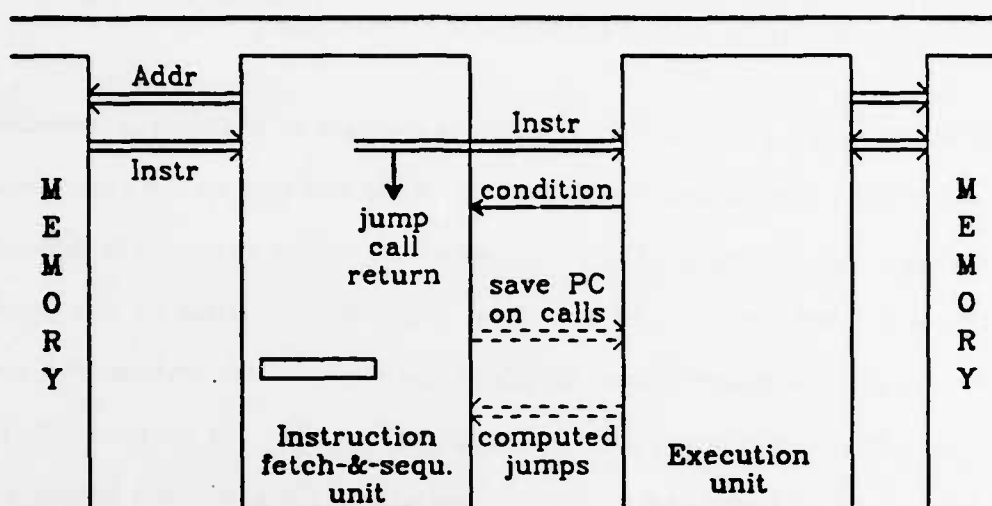


Figure 6.3.1: Remote-PC Scheme.

two possible paths. For unconditional sequencing, a single path is possible, whether it be a linear (no transfers), or a non-linear (unconditional transfer) one.

- *Static/Dynamic Transfer Target:* The address of the instruction to-be-executed-next is usually known statically at compile time and thus does not depend on the execution-unit. Exceptions are the *return* instructions and the infrequent "computed jumps" (e.g. for *case* statements).

The average communication bandwidth between a remote-PC instruction-unit and the execution-unit is not much more than the minimum bandwidth needed to merely supply the instructions to the execution unit. That is so because addresses are transmitted infrequently between the two units. Control transfers with a dynamic target are not very frequent (return instructions are less than 5% of all executed RISC instructions [PaSe81]). Also, the jump instructions do not need to be sent to the execution unit. The bandwidth achieved in this way is significantly lower than the one required in the conventional scheme, where the *PC* is kept in the execution unit and has to be sent out of it for every single instruction-fetch. This low bandwidth shows that information is maintained and processed locally to a maximum degree; this makes such a partitioning desirable.

In order for the remote-PC implementation to be successful, the value of *PC* should be used as little as possible in the execution of instructions. In particular, no PC-relative addressing mode should exist for data accesses (see § 3.1.2). An instruction-unit with remote-PC and with an instruction cache, will be the basis for the hardware enhancements proposed in the rest of this section. The Instruction-Cache chip that was designed and built for RISC II [Patt83] does include a remote-PC; however, the latter contains only an estimated value of the real *PC*, and is used for predictive fetching of instructions. The RISC II CPU was not designed for a remote-PC system, and thus it includes the *PC* in itself, and it has PC-relative *load* and *store* instructions.

### 6.3.2 Jumps, and Delays Introduced by them.

An instruction fetch-&-sequence unit has to deal efficiently with control-transfer instructions, because they occur very frequently. The following table reviews some of the measurements presented in § 2.2.1:

Property:	Measurement:	Reference:
<i>Opcodes, dynamically:</i>		
branching instructions	30 %	[Lund77]
branch instruction	14 %	[AlWo75]
uncond. jumps, rel. to all jumps	55 %	[AlWo75]
<i>HLL statements statically:</i>		
if	13 %	[AlWo75]
call	13 %	[AlWo75]
<i>HLL statements dynamically:</i>		
if	38 % ± 15 %	[PaSe82]
call	14 % ± 4 %	[PaSe82]
loops	4 % ± 3 %	[PaSe82]
call	12 %	[Tane78]

These numbers show the validity of the commonly used rule of thumb that "one out of four executed instructions is a control-transfer". The following program fragment out of § 2.4.4 is quite typical of non-numeric programs, as far as

frequency of jumps is concerned, and illustrates the same point:

```

while(new != NIL && old != NIL)      /* NIL is 0 */
{
  if(new->bb.l < old->bb.l) { infrequent }
  else { if(n < old->bb.t)
        { if(last == NIL) { rare }
          else { last->next = old; last = old; }
          old = oldList;
          if(old != NIL) oldList = old->next;
        }
        else { infrequent }
      }
  if(depth[last->layer] == 0) { 50% of the times: call a procedure }
  if((depth[last->layer] += last->dir) == 0) { again 50%: call }
  nextEnd = (nextEnd < last->bb.t ? nextEnd : last->bb.t);
}

```

Besides illustrating the high frequency of *jump* instructions, the above program fragment also shows the intimate connection between *jumps* and *test* or *compare* instructions. The usual pattern is that a number is compared to zero (*test*), or two numbers are compared to each other (*compare*), and a conditional jump is then executed, based on the outcome of the comparison. Hennessy et.al. studied how many of the conditional jumps require an explicit comparison operation performed for their sake and how many of them can use the result of some other instruction [Henn82, table 2.3]. They measured that less than 2 % of the conditional jumps were able to use condition-codes set by an instruction that was *not* executed solely for that purpose.

Fast control-transfer instructions are particularly important for high processor performance, because they are so frequent and because they block the fetch-execute pipeline. This importance becomes even more significant when the combined time spent for branches and comparisons is considered. For that reason, the rest of this section will focus on fast *compare-and-branch* operations.

### 6.3.3 Fast Compare-and-Branch Scheme.

Figure 6.3.2(a) shows the compare-and-branch scheme followed in RISC I & II. Given that in about half of the cases the optimizer is able to move something useful into the cycle labeled "OTHER", we can say that this scheme takes about 2.5 cycles, on average, for a comparison and a branch.

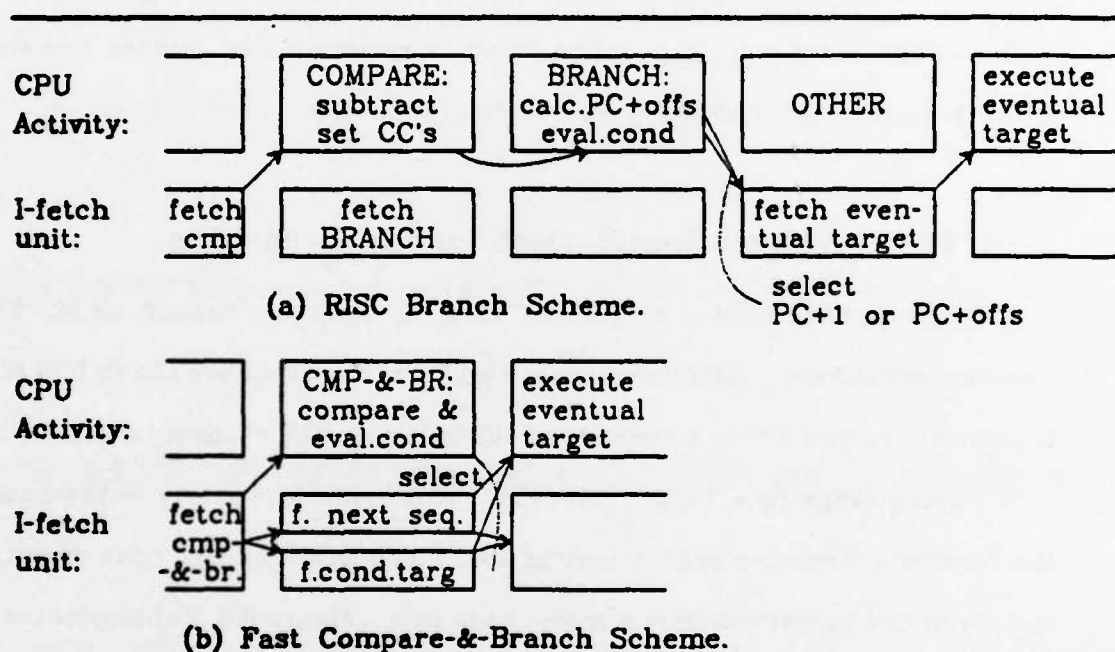


Figure 6.3.2: Branching Schemes.

There are two possibilities for improvement of this scheme. First, the comparison and the decision whether to take the branch or not can be executed in parallel with the computation of the possible branch target,  $PC + \text{offset}$ . This would require two ALU's, but it would reduce the time for a compare-and-branch to about 1.5 cycles. This scheme comes only natural when a separate instruction fetch-and-sequence unit is used, like the one of § 6.3.1. Second, the branch target address is known at compile time, and there is no reason — other than code compactness — why it should be recomputed every time the branch is executed. When an instruction cache is used, a solution exists that allows both the

code to be compact and the target address computation not to slow down the instruction-fetching. It will be presented in the next subsection 6.3.4.

Figure 6.3.2(b) shows the proposed fast compare-&-branch scheme. It makes use of both of the above improvements, and it allows single-cycle compare-&-branch instructions. However, now there is not enough time for the jump/no-jump decision to be made before the fetching of the target instruction begins. Thus, a two-port instruction cache is required, that fetches simultaneously both possible targets of the conditional branch.

#### 6.3.4 Target Address Specification for Fast-Branching.

Most branches have a target not very far from the branch itself. Some measurements for a particular language and architecture have shown 55% of the branches targeted within a distance of 128 bytes, or 93% of them targeted within a 16 Kbytes range (see § 2.2.1 [AlWo75]). This locality property is the basis of the familiar PC-relative branch instruction, which achieves high code density by specifying the target's *distance* from the branch. Figure 6.3.3(a) illustrates this method. In that figure, thick lines represent information statically determined (by the compiler) and thus included in the instruction. Thin lines represent dynamically computed information. In the traditional PC-relative branch scheme, shown in (a), the instruction contains a  $(n+1)$ -bit *offset* field, which is added to the *PC* at execution time and produces the conditional target address. All branches to within a distance of  $\pm 2^n$  from the current instruction can be represented with this instruction format (see the little graph on the right).

The rest of figure 6.3.3 shows three variants of the proposed alternative fast-branching scheme. The key idea here is that the instruction contains the  $n$  least-significant (LS) bits of the *conditional target address* itself, rather than of its *offset*. In this way, the instruction cache can start fetching that target as





compact branch instruction that cannot contain the whole address. However, this computation can be performed in parallel with the cache RAM access, as long as its result becomes available in time for the address tag comparison. There are three different ways in which the MS part of the conditional target address can be computed in the fast-branch scheme, and they are illustrated in parts (b), (c), and (d) of fig. 6.3.3.

The straightforward transformation of the traditional scheme (a) into the fast-branch scheme is shown in (b). The sign-bit of the *offset* and the carry-bit from the (virtual)  $n$ -LS-bit addition are computed by the compiler and included in the instruction, so that the MS part of the same addition can be recreated. This scheme requires an  $(n+2)$ -bit instruction-field for specifying the conditional target, and it achieves a worst-case branching range of  $\pm 2^n$ .

The second variant shown in (c) is better than that of (b). The compiler supplies again 2 bits of information for computing the MS part of the conditional target address. However, instead of adding *both* of them to bit  $\langle n \rangle$  of *PC* as in (b), they are considered as forming a 2-bit signed number which is added to the MS part of *PC*. This achieves a worst-case branching range of  $-2^{n+1}$  to  $+2^n$ . The scheme in (d) is similar to that in (c). It uses the  $PC\langle n-1 \rangle$  bit as a round-off bit to achieve an equally balanced worst-case branching range of  $\pm 1.5 \times 2^n$ .

Figure 6.3.4 shows the block diagram of an instruction fetch-and-sequence unit that incorporates a two-port instruction cache and the fast compare-and-branch scheme. (This is the simple form of the I-unit; figure 6.3.6 shows the full form.) The double register  $\{PCplus\ 1, \text{Instruction Register}\}$  at the CPU interface is loaded at the beginning of each execution cycle with the instruction to be executed and with its incremented *PC* value. The incremented-*PC* value is used as the address for one of the two instruction-cache ports, and causes the subsequent instruction to be fetched. Simultaneously, the appropriate field of the

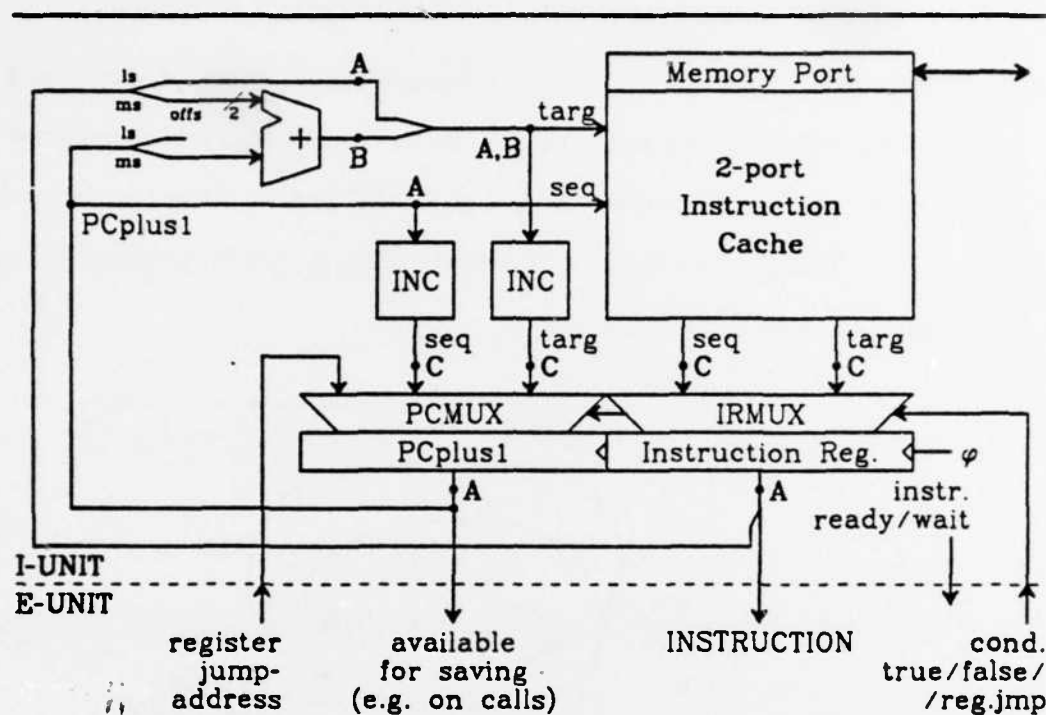


Figure 6.3.4: I-fetch Unit with Fast-Branch Scheme (simple version).

current instruction — assuming that this is a branch — is used for determining a possible target-address, according to the scheme of fig. 6.3.3 (c) or (d). This possible target-address is fed to the second port of the instruction cache, and a possible target-instruction is fetched. At the end of the cycle, the execution unit has decided whether this was a conditional branch, and whether it should be taken or not. According to that decision, the multiplexors IRMUX and PCMUX select the output of the first or second cache port and the output of the first or second address incrementer, in order to load the instruction- and the incremented-PC registers. Notice that the instruction cache should *not* initiate the miss-process before it is certain that the offending access is for an instruction that will actually be executed. The next subsection looks at the timing of compare-&-branch instructions in more detail.

### 6.3.5 Form of Comparisons in the Fast-Branch Scheme.

Figure 6.3.5 is a graph of the timing-dependencies for the compare-&-branch instruction. It is similar to the one in figure 4.2.1, and it assumes a processor with a data-path similar to that of RISC II and with the fast-branch I-unit of fig. 6.3.4. The points labeled A, B, and C on this graph correspond to the similarly labeled points in fig. 6.3.4.

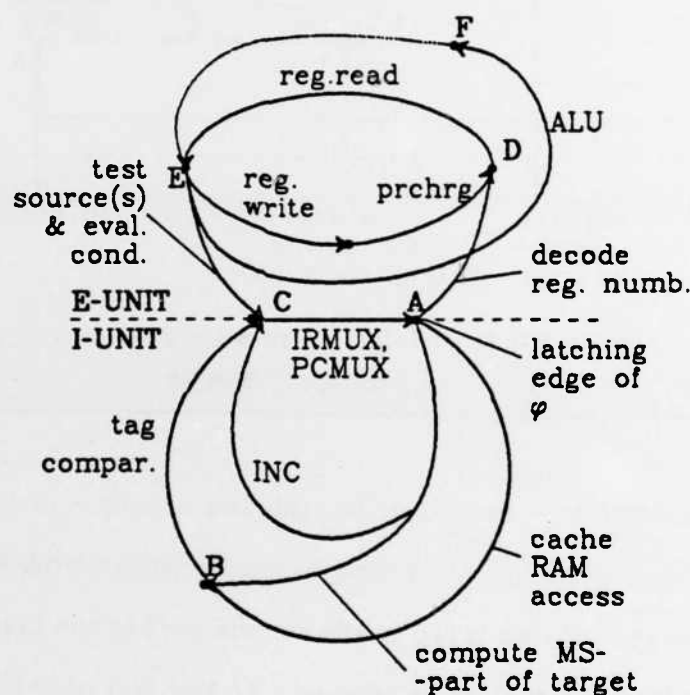


Figure 6.3.5: Timing Dependencies in Proposed Compare-&-Branch Scheme.

At point A, an instruction is ready to start executing. Assume that it is a compare-&-branch instruction that has to perform a comparison in the execution-unit (upper cycle), while the I-unit fetches its two possible successor instructions (lower cycle). The  $n$  LS bits of the addresses of both candidate successors are known at point A (see figures 6.3.4 and 6.3.3(c,d)). Thus, the cache RAM access can begin immediately. (We assume that  $n$  bits are enough to

address the words and the blocks of the cache RAM). The cache RAM access is complete at point B, at which point the MS part of the conditional-target must also have been computed. The cache tag comparison may begin at point B and must be completed at point C when the next instruction is ready to be selected. At the same point C, the two incrementers INC (fig. 6.3.4) must have valid outputs; notice that the carry of the second INC propagates in parallel with the carry of the target-address addition.

At the same point C, the execution-unit must have decided whether the branch should be taken or not, so that the next instruction can be selected. Assuming an instruction set and a data-path similar to those of RISC II, the compare-&-branch instruction must decode two registers (A→D), must read them from the register-file (D→E), and must compare them and decide (E→C). This comparison may or may not require a subtraction. Figure 6.3.5 clearly shows that it would be overly restrictive for the data-path to leave enough time between points E and C for a full-width subtraction. The reason is that all other instructions may allow almost a full cycle E→F for the ALU operation. (See fig. 4.2.1; however, here, we do *not* want the ALU to be on the I-fetch critical path.) Thus, the possible forms of the comparison should be restricted, so that *no subtraction* is required.

This means that comparisons to zero ("tests") can be allowed, as well as comparisons of two arbitrary quantities for equality or inequality. None of those requires a circuit with carry-propagation for its detection — they are resolved by just looking at the most significant bit of the source or by using a wide precharged NOR gate to check for equality to zero, possibly after computing a bitwise exclusive-OR in the ALU. However, a magnitude comparison of two arbitrary quantities should not be allowed in a compare-&-branch instruction, since it requires an adder or a priority encoder for its implementation.

This restricted comparison form is another example of an architectural decision made for implementation reasons. To evaluate its effects in real programs, the critical program fragments presented in sections 2.3 and 2.4 were analyzed by hand.

All comparisons for conditional branches were classified into four categories:

- TST: arbitrary comparisons to zero;
- EQ/NE: comparison of two quantities for equality/inequality;
- CMP-SOFT: those comparisons used for deciding the termination of a DO-loop, which are of the form *if* ( $i \geq LIMIT$ ), and which could be re-written in the form *if* ( $i == LIMIT$ );
- CMP-HARD: all comparisons of two arbitrary quantities for  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ , and which are not CMP-SOFT.

They were counted dynamically; the average percentages in each type of comparison are given below, separately for the 17 numeric critical loops (§ 2.3), and for the 3 non-numeric programs (*fgrep*, *sed*, *mextra*: § 2.4):

	TST	EQ/NE	CMP-SOFT	CMP-HARD
numeric programs	18 % $\pm$ 32	13 % $\pm$ 23	51 % $\pm$ 43	18 % $\pm$ 28
non-num. programs	55 % $\pm$ 11	24 % $\pm$ 8	6 % $\pm$ 8	15 % $\pm$ 15

These numbers show that, with no program re-writing, the restricted-form comparisons would be useful in about 40% of the cases for numeric programs and in about 80% of the cases for non-numeric ones. With program re-writing or with language semantics that allow equality comparison in DO-loops, these numbers would become about 80% and about 85%, respectively. Thus, the restricted comparison form appears quite frequently in real programs, especially in non-numeric ones.

From the point of view of the instruction format, the compare-&-branch instruction can fit in 32 bits, although certainly not in a fashion compatible with the RISC I & II instruction format. This incompatibility may result in performance penalties due to a more complicated instruction decoding, if this scheme is used within an ISP like that of the Berkeley RISC. A possible set of

instruction-fields and widths for that instruction is the following:

- 3-bit opcode,
- 4-bit branch-condition specifier and  $S2$  selector,
- 5-bit  $R_{s1}$  specifier,
- 8-bit  $S2$ :  $R_{s2}$  or a byte-wide immediate,
- 12-bit target-address specifier ( $n=10$  in fig. 6.3.3).

The 12-bit target-address specifier allows the use of 10-bit addresses for the cache RAM. This corresponds to a maximum cache size of 2 K-instructions for a 2-way associative cache, or 4 K-instructions for a 4-way associative one, which are reasonable limits.

### 6.3.6 Extension for Zero-Delay Unconditional Branches.

Figure 6.3.6 shows the full version of an instruction fetch-and-sequence unit with a two-port instruction cache. This version, besides implementing the fast compare-&-branch scheme of § 6.3.3, also executes unconditional branches in "zero-time", i.e. without holding the execution-unit while it follows those branches. An exception are unconditional branches which follow conditional ones within a distance of 1 or 2 instructions: they *will* hold the execution unit for 1 cycle.

As we saw in § 6.3.2, about half of all branches are unconditional, thus accounting for roughly 1/10 of all executed instructions. They arise in a natural way when the non-linear flow-diagram of a program is converted into machine-code stored in a linear memory. Unconditional branches describe no useful computation; they just divert the instruction-fetching process onto a different path. It is interesting to note that unconditional branches are usually "for free" in micro-code since micro-instructions usually contain the address of their successor inside themselves. The same general scheme can also be used with macro instructions.



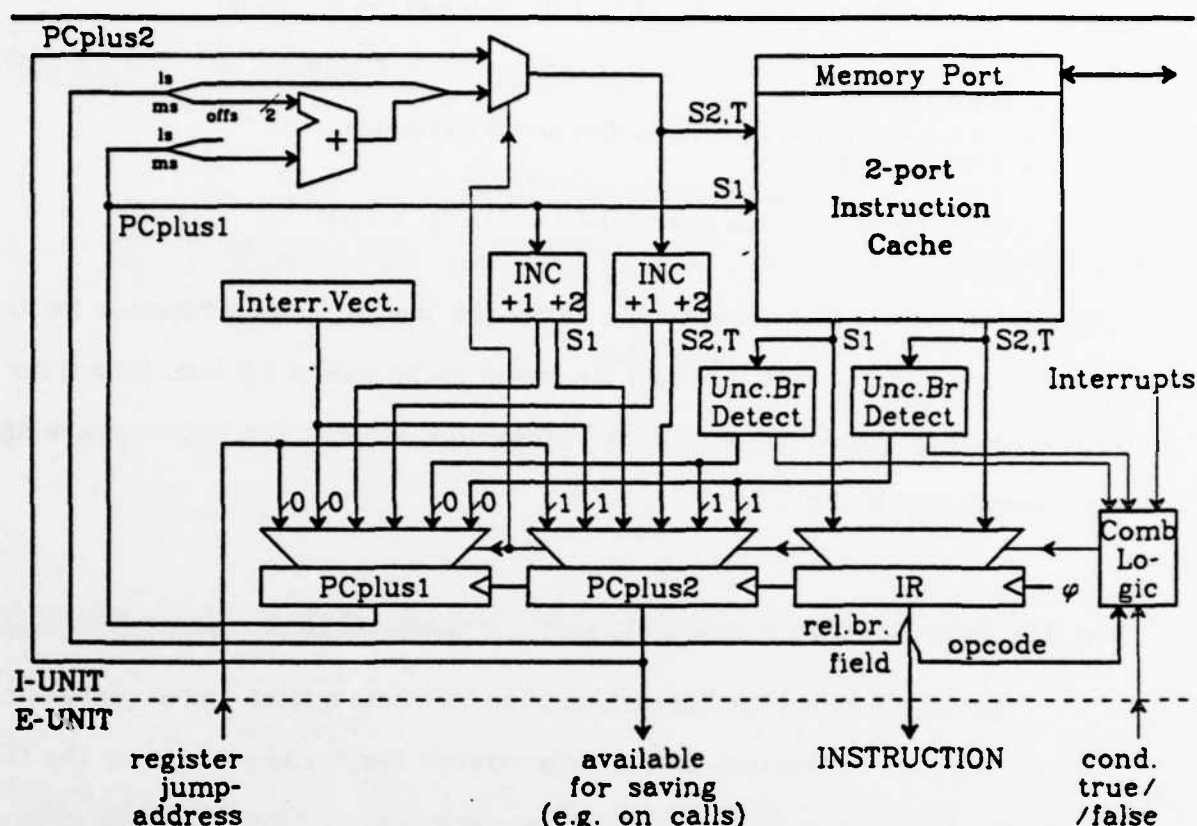


Figure 6.3.6: I-fetch Unit with Fast-Branch (full version).

A two-port instruction cache is used for simultaneously fetching both possible targets of compare-&-branch instructions. The basic idea behind the full version of the I-unit is to also exploit both ports of the instruction-cache when instructions other than compare-&-branch are executing. When such an "other" instruction is executing, the *PCplus1* and *PCplus2* registers are used to fetch its next two instructions, say  $I_{S1}$  and  $I_{S2}$ . If  $I_{S2}$  is an unconditional branch, and  $I_{S1}$  is neither an unconditional nor a conditional one, then  $I_{S1}$  can be supplied to the E-unit for normal execution, and the target address of  $I_{S2}$  can be followed *immediately*. While  $I_{S1}$  is being executed in the E-unit, the *target* of  $I_{S2}$  is fetched; and the unconditional branch becomes invisible to the execution unit.

Unconditional branches can have an instruction-format different from that of conditional ones. This makes possible the immediate pursuit of the target address without the need for techniques similar to those of the fast compare-&-branch scheme (§ 6.3.4). Unconditional branches can have a 3-bit opcode and a 29-bit absolute target address. It is advantageous to make the target of all unconditional branches be an even-word aligned instruction. In that way, both the target instruction and the instruction next to the target can be fetched by concatenating the 29-bit target-field with "000" and with "100" (byte addresses), respectively. These quantities are fed to  $PCplus\ 1$  and  $PCplus\ 2$  as soon as  $I_{S2}$  is detected to be an unconditional branch. This format for unconditional branches also offers a means of branching to arbitrarily distant memory locations, unlike the restricted-range compare-&-branch instructions.

*Call* instructions perform the same function as do unconditional branches do, except that they must also save the  $PC$  for use upon procedure return. If the place to save the  $PC$  is a fixed register in the child's window, then *call* instructions can have the same format of a 3-bit opcode and a 29-bit absolute target address. The I-unit can treat *calls* similar to unconditional branches, except that it must also supply them to the E-unit, which must execute them by saving the  $PC$ .

Normally, an unconditional branch should first appear as the second one of the two consecutive instructions  $I_{S1}$  and  $I_{S2}$  being fetched from memory locations  $M[PCplus\ 1]$  and  $M[PCplus\ 2]$ , respectively. The reason is that, if execution has been sequential in the recent past, then an unconditional branch fetched via  $M[PCplus\ 1]$  would also have been fetched via  $M[PCplus\ 2]$  one cycle earlier and would have been executed at that time. That will not happen if execution has not been sequential in the recent past, and specifically if an unconditional branch is the target of another one or if it follows a compare-&-branch within a distance

of 1 or 2 instructions. (Notice that occurrences of the former situation can be removed by the optimizer or the linkage editor.) Because these cases have to be dealt with, unconditional branches should also be detected and handled when appearing at the first (*S1*) port of the I-cache. However, in those cases there is nothing else useful that can be supplied to the E-unit while the target of the branch is being fetched. Thus, the unconditional branch itself can be given to the E-unit, which should interpret it as a *noop*.

In this section, instruction fetch-and-sequence units of increasing sophistication have been proposed. When sufficient hardware resources are available for the implementation of such an I-unit, a high-performance execution-unit can be kept busy and the time spent executing control-transfer instructions can be reduced. Roughly 1.5 cycles per conditional branch and one cycle per unconditional branch can be saved, amounting to about 2.5 cycles out of every 10 cycles of execution.

## 6.4 Pointers and Data Caches.

According to the list of proposed priorities at the beginning of this chapter, hardware resources that remain available after a multi-window register file and an instruction-unit have been implemented, should be spent for a data cache. The two former devices were investigated in the previous sections. In this section, the special nature of accesses to non-scalar data is considered as far as the construction of an effective data cache is concerned, and hardware as well as programming methods for its exploitation are proposed.

### 6.4.1 Data-Structure Accesses and Data Caches.

Operands used by programs are either scalar variables or elements of non-scalar data-structures. These two categories differ fundamentally, as pointed out in § 8.1.1. Scalars are few in number, they occupy little memory space, they are referenced using their own name, and several of them are used repeatedly. They are often used to refer to particular elements of data-structures. Data-structures, on the other hand, have many elements and occupy large memory space. Individual elements of these structures are accessed via dynamically computed addresses.

Accesses to non-scalars follow certain typical patterns:

- A • A number of *repeated accesses to the same element* is often made before interest shifts to another element of the data-structure. For example,  $A[i,j]$  is accessed three times during each critical-loop iteration in fig. 2.3.1; the element *last*  $\rightarrow$  *layer* is accessed  $\approx 4$  times per iteration in the procedure *ScanSubSwath()* in § 2.4.4. Another, less frequent, case arises in the critical loop of *fgrep* in fig. 2.4.1. The pointer *c* is pointing to the same element of the structure *words* during most loop iterations, because the scanner is searching for the same first letter of the desired pattern most of the time.
- B • *Accesses to near-by memory locations* are frequently made. They arise in two different ways. First, arrays are frequently traversed in a sequential manner such that each element accessed is next to the previously visited one, in terms of its memory address. This is true for sequential scanning of linear arrays (character buffer scanning is a common case), as well as for the scanning of multi-dimensional arrays *by columns* (in FORTRAN). These occur quite frequently (see § 2.3, 2.4.1, 2.4.2). Second, more than one of the fields of a structure are usually accessed before program execution moves to another structure (§ 2.4.4). Since structures are often small in size, their fields are in near-by memory locations. To evaluate that size, static measurements were collected (by hand) on the size of the structure types declared in 15 C programs, including a screen editor (emacs), a HLL interpreter (logo), *fgrep*, *sed*, and *maxtra*. Among 54 structure declarations investigated, about 45% of them were found to have 4 or less words; about 70% had 8 or less words; and about 85% had 16 or less words. All sizes are for the VAX-11/780, where 1 word equals 4 bytes.
- C • However, the occasional *shift of accesses to remote locations* can not be neglected. It occurs whenever array accesses are not sequential in address space, or when various nodes of dynamically allocated data-structured are accessed. The former case is not very frequent for linear arrays, but it is common for multi-dimensional ones. The latter case occurs due to the "random" allocation in memory of nodes that are linked to each other. In

practical situations, however, this allocation is not completely random, and some linked nodes do end up next to each other. A study of 5 large LISP programs by Clark and Green [ClGr77], for example, showed that about 1/4 of the *car* and *cdr* list pointers were pointing to the immediately adjacent (forward direction) memory cell.

Registers are the natural device for holding frequently used scalars, because of the small number and size of those variables. A cache memory, on the other hand, is well suited for keeping the elements of data structures, because of access patterns A and B above. The fact that caches keep the most recently used words in them accelerates type-A accesses, while the fact that caches fetch a whole block when a word is missed accelerates type-B accesses.

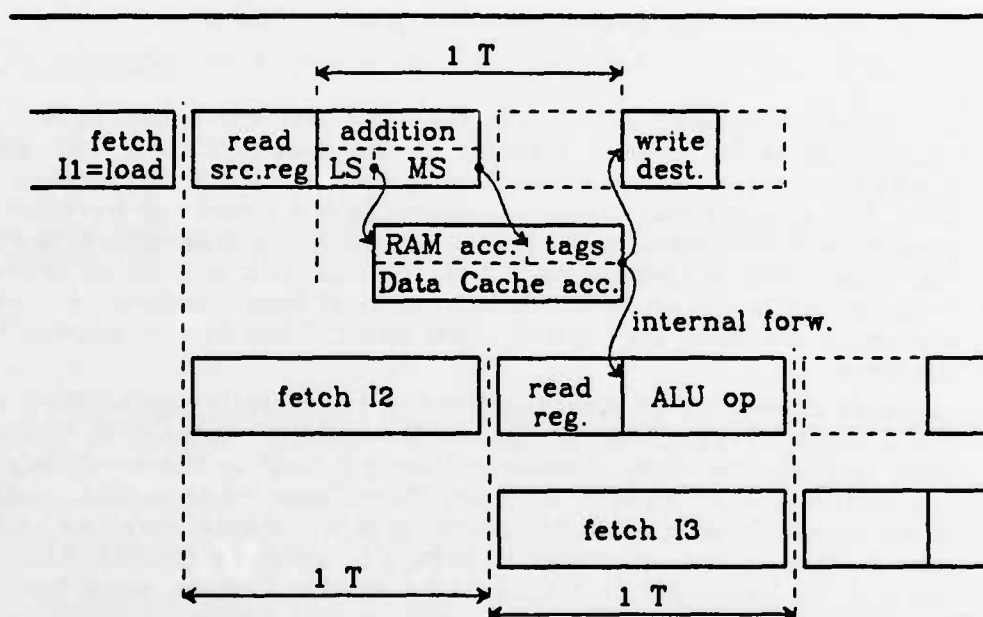


Figure 6.4.1: One-Cycle Load Instructions in a RISC II with a Data-Cache.

A data-cache, that is separate from the instruction-fetch port of the CPU, can allow one-cycle *load* and *store* instructions in the RISC II architecture and pipeline. A timing scheme similar to that of fig. 3.3.4(b) is used for that purpose, except that here no restriction needs to be placed on the RISC II addressing mode  $R_s \leftarrow M[R_{s1} + S2]$ . Figure 6.4.1 shows the timing of a *load* instruction

when a data-cache is used. The data-cache access may begin just a little while after the addressing addition has started in the ALU. The reason is that only the least-significant bits of the effective address  $R_{s1} + S2$  are required for the cache RAM access. The rest of the bits of the effective address are needed later on when the address tag comparison takes place. Internal forwarding allows the next instruction  $I2$  to use the loaded data. For a cache that requires the  $n$  LS address bits for its RAM access, the timing constraint is:

$$(\text{Data Cache Access Time}) \leq T - (n\text{-bit Add Time}).$$

Notice that for this scheme to be possible, the data-cache needs to be separate from the instruction-fetch port of the CPU (e.g. the instruction-cache), so that the required parallel access to both of them is possible.

Cache misses limit overall performance. Accesses of type C cause misses, and so do those of type B when they cross block boundaries. The next subsection proposes hardware and programming methods for reducing the number of those misses.

### 6.4.2 Pointers as Pre-Fetching Hints.

There is an intimate connection between pointers and data-structure accesses. Most accesses to fields of structures are made indirectly through a pointer scalar variable that is pointing to the structure, and most of those pointer variables are local ones. Similarly, in non-numeric programs, array accesses are often made using pointers to the array elements such as character-buffer accesses in text-processing programs. For example, 85 % of the accesses to non-scalar data made in the critical loops in § 2.4 have the form  $*p$  or  $p \rightarrow fld$ , where  $p$  is a local scalar variable; the remaining 15 % of those accesses have the forms  $arr[n]$  or  $(non\text{-}scalar) \rightarrow fld$  †. In numeric programs,

† These measurements are static; however, they were collected only from critical loops (§ 2.4).



array accesses are traditionally made using subscripts (in languages like FORTRAN this is the only choice). These, however, could also be replaced by indirections through local pointers by the optimizer or by the sophisticated programmer, as noted in § 2.3.4. Such a replacement would simplify the required address computations, and would also make forthcoming proposals applicable to these accesses as well.

When a scalar pointer variable is loaded with a new value, it is very likely for that value to be used shortly afterwards for an indirect access to an array element or to a field of a structure. For example, after 90 % of the assignments to a local pointer  $p$  made in the critical loops in § 2.4, the assigned value was used for an access of the type  $*p$  or  $p \rightarrow fld$ . These accesses are always made to the memory word where  $p$  is pointing to ( $*p$ ), or to neighbouring words ( $p \rightarrow fld$ ). In the remaining 10 % of the cases,  $p$  was used for purposes other than indirect memory accesses – for example it was compared to a limit value ‡.

This suggests that a data cache should use the assignment of a value to a pointer variable as a *hint for prefetching* into the cache the block where this pointer is pointing to, if it is not already there. In this way, some of the type-C accesses (§ 6.4.1) may be turned into type-A or type-B ones, and the miss ratio may be reduced. In a processor where registers are used to hold the most frequently used scalar variables, the criterion for prefetching a block into the data cache may simply be the writing of a pointer value into a register. One way of distinguishing pointer from non-pointer assignments is by setting aside some registers for pointer values only ("index registers"). This, however, reduces the flexibility and orthogonality of the architecture and leads to non-optimal utilization of the register file. A better way is to have the compiler tell the machine,

‡ These measurements are static; however, they were collected only from critical loops (§ 2.4).



with one bit in the instruction, that the assigned value is a pointer one. The proposed prefetching scheme is reminiscent of the way loads and stores are performed on the CDC-6600 computer [Thor64]; a memory-to-data-register transaction is implicitly initiated every time an address-register is loaded with a value.

The success of this prefetching scheme in actually reducing cache misses is critically dependent on the amount of time available between the assignment of a value to a pointer and the first use of that value for an indirect memory access. That time interval has to be long enough for the corresponding block to be prefetched into the cache before an access is first made into the block. Depending on the system organization, the time to fetch a block into the cache may vary in the range of about 4 to 10 machine cycles, or about 4 to 10 RISC-style instructions. Of course, even if that time is not available between the assignment to a pointer and its first use, a gain still exists in the form of a shorter miss delay. To get an estimate of the above time interval, we counted the HLL statements in the critical loops in § 2.4 that are executed between loading a pointer and first indirecting through it †. The results were as follows:

- ≈ 50% of the cases (8 pointer loadings): 0 or 1 HLL statements
- ≈ 20% of the cases (3 pointer loadings): 2 or 3 HLL statements
- ≈ 30% of the cases (4 pointer loadings): 4 or more HLL statements

This means that only in 30 to 50 % of the cases there is enough time for the prefetch to be complete before the first indirection through the pointer actually occurs.

It is possible for this time interval to be lengthened, and thus for the prefetching-hint scheme to yield better results, through a more *sophisticated*

† These measurements are static; however, they were collected only from critical loops (§ 2.4).

*programming technique.* In critical loops, the programmer can *preload* a local pointer with an address to be (potentially) used during the next loop iteration.

For example:

```
/* Run through a list, doing some processing with its elements */
struct node *current, *next;      /* local pointers */
next = head;
while ( (current=next) != NIL )
    { next = current->nxt;      /* preload */
      ...Do the processing, using current->otherFields...
    }
```

In a few cases, a code rearrangement with similar effects could be made by an optimizing compiler. However, most cases are such that the prefetching will only be effective when done almost one whole loop iteration ahead of time. This usually requires the introduction of a new pointer variable by the programmer.

Even with this sophisticated programming technique, misses will still occur whenever a pointer to a structure is loaded with a value pointing into one cache block and is subsequently used to access a field of the corresponding structure which overflows onto the next cache block. This will happen most frequently with pointers pointing near the end of blocks or with structures that are larger than the block size. One solution can be to prefetch both the block pointed to by *p* and the next block whenever *p* is loaded with a value that points "too close" to a block's end. Another solution can be to try to allocate structures so that they do not cross cache block boundaries too often. Assuming a block size of 8 words = 32 bytes, this can be done trivially for structures of sizes 2, 4, or 8 words. In the static measurements reported in § 6.4.1 (B), the structure declarations that had exactly those sizes constituted 10%, 25%, and 10% of all structure declarations, respectively.

## 6.5 Multi-Port Memory Organization.

Throughout this dissertation our focus has been the central processing unit of a von Neumann computer. The predominant pattern of simple operations applied to large volumes of operands was observed in typical examples of frequently occurring computations. As a consequence, priority was given to hardware support for fast operand accesses in the forms of multi-window register files, instruction fetch-and-sequence units, and data caches. However, the prevalent role of access to data or information is not confined to the interaction of the CPU with its surrounding fast storage devices. Also important is the access to bulk storage devices, to other processors, and to remote computing sites. This section concerns itself with the access to information in main memory. The use of memories with multiple ports will first be established. Then, a new device is proposed, a modified dynamic RAM chip, which effectively provides a second, independent, *sequential-access* port at a minor additional cost.

### 6.5.1 The Need for Multi-Port Memory Systems.

Figure 6.5.1 presents an overview of various system organizations, showing their requirements for multi-port memory systems.

- (a) is a uni-processor, perhaps with a single cache memory. Slow I/O accesses (e.g. terminals, telephone lines) may be made via the CPU or via memory, but fast I/O accesses (e.g. disks, local-area-networks (LAN), raster displays) are always made by direct-memory-access (DMA) since their bandwidth is so high that the CPU cannot handle it. At least two high-bandwidth memory ports are seen to be needed: CPU and fast-I/O.
- (b) is a higher performance uni-processor with two memory-ports, one for instructions and one for data (§ 3.3.2, 3.3.3). This system may have separate instruction and data caches, as proposed in this chapter, but simultaneous misses in both caches are possible and require a memory port for each one of them. Such systems can be implemented with two separate main memory modules, one for instructions and one for data.

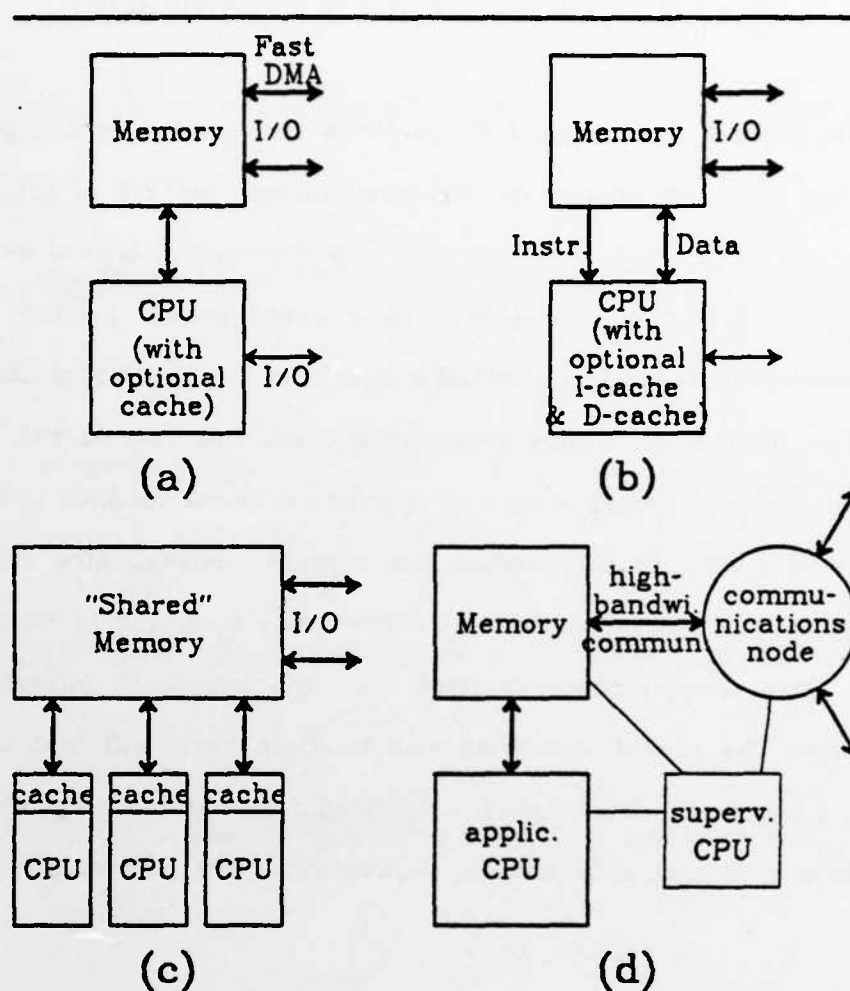


Figure 6.5.1: Need for Multi-Port Memories.

However, it is also common to implement them with a single shared module, so that all available memory can be used regardless of whether there is little code and many data or vice versa.

- (c) is one possible multi-processor configuration. Here, the need for multiple memory ports is very high. Accessibility to the memory is, in fact, the limiting factor in system expandability.
- (d) is a possible organization of one node in a future multi-processor system. It consists of a uni-processor like (a) or (b), connected to other processors and I/O devices via a high-bandwidth network of interconnected communications components [Fuji83]. A separate supervisor CPU may exist, to take care of communications overhead, paging, interrupt handling, etc. We consider this organization as more attractive than (c), because it is expandable in a uniform manner.

Today, multi-port memories are made by providing time-shared access to a single physically available memory port. This is because true multi-port devices are prohibitively expensive for large storage systems, but also because most systems are of type (a) above with fast-I/O devices that have an average bandwidth that is still significantly lower than that of the CPU.

When multiple, asynchronous access requests are made to a single, shared memory port, contention will arise. Whenever simultaneous requests occur, all but one of the requesting devices will have to wait. For example, a RISC CPU uses the memory during all cycles; thus, it has to wait when an I/O access occurs. In a VAX-11/780 system, the processor is slowed by roughly 4% for each Mbyte/sec of average I/O traffic when its memory is 2-way interleaved, or by roughly 10% in the non-interleaved option [DEC81].

If the average bandwidth of the fast-I/O devices is significantly lower than that of the CPU -- and hence than that of the memory -- then memory contention will only have a minor negative effect on CPU performance. The following table gives a feeling of the bandwidth requirements of the various devices that may be connected to a memory system in organizations like the ones in fig. 6.5.1. Since real values vary widely, the table only gives some typical examples.

<i>Example of Memory Access Characteristics by Various Devices.</i>			
Device	Size of access (Bytes)	Average Bandwidth (MBytes/sec)	Peak Bandwidth (Mbytes/sec)
CPU	4	10.	12.
Cache	32	4.	20.
Slow I/O (Terminal, etc)	1	0.0001	0.001
Fast I/O (Disk, LAN)	2000	0.3	1.

Thus, in typical systems of today, the average bandwidth of their few fast-I/O devices represents only about one tenth of the CPU bandwidth. This is a tolerable load which does not degrade CPU performance too much, when the CPU and

I/O share a single memory port with a cycle-time approximately equal to the CPU cycle-time.

This situation, however, will probably change in the future. In our view, systems will evolve towards organization (d), since expandable multi-processors will be desirable and feasible. The communications bandwidth will increase, both because of advances in technology, and because of the need for more closely-coupled parallel execution. This bandwidth increase will place a heavy load on the memory system. It is important to notice, however, that, whereas the CPU performs random accesses to the memory, a fast I/O device or a communications network performs *serial* accesses most of the time, since large pieces of consecutive memory (e.g. pages) are copied in or out of the memory. The internal organization of dynamic RAM chips is such that it allows the inexpensive addition of a second *serial-access* port to them, thus solving the CPU-I/O contention problem.

### 6.5.2 DRAM Chips With Secondary Serial-Access Port.

The top part of figure 6.5.2 shows a typical internal organization of a dynamic RAM chip. Every time an access is made, the whole row along the activated word-line is read out of the storage array into the sense amplifiers. Then, a single bit is selected by the column address. Thus, there is a huge difference of about 3 orders of magnitude in the bandwidth of on-chip and off-chip reading operations. In a memory system, the chips' row-address is typically the physical page number. Thus, on every access, one whole memory page is read into the sense amplifiers of the memory chips. For example, if a 1 Mbyte memory consists of 32 256-Kbit chips like the one shown in fig. 6.5.2, then, on every memory access, 32 rows of 1-Kbit each are read in all the chips, amounting to a total of a 32-Kbit or a 4-Kbyte page.



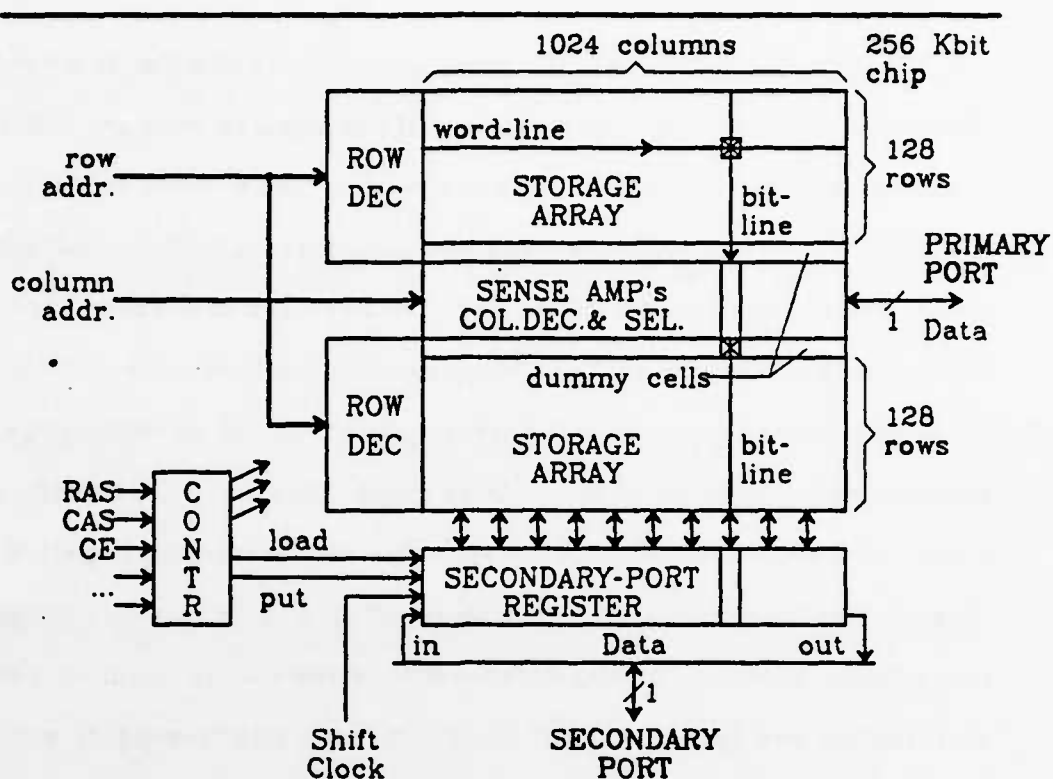


Figure 6.5.2: DRAM Chip with Secondary Serial Port.

Even though a whole page may be read inside the DRAM chips, only a single word is selected out of it and transmitted off-chip. Many recent DRAM chips offer a *nibble* or *page* mode of access. Under the former mode, four adjacent bits (words) out of the row (page) are serially transmitted off-chip, with a very short delay between subsequent bits. Under the latter mode, any subsequent access to the same row (page) can be made with a shorter delay than the first one, by just supplying a new column-address to the chips. These modes of access make some of the high on-chip bandwidth available to off-chip accesses. They can be used to speed up cache-block accesses and sequential I/O as long as it is uninterrupted by references to other pages. The applicability of these modes to CPU accesses is limited because the CPU does not usually ask for adjacent words or for words from the same page on consecutive memory



transactions.

The lower part of figure 6.5.2, shows the proposed addition to a conventional DRAM chip. It consists of a shift-register of size equal to one row. It has parallel connections to the bit lines of the storage array and a serial connection to one pin of the chip. One additional chip pin is used as its *independent* shift clock. This proposed "secondary-port register" can be loaded in one memory cycle with the contents of the row which is being read during that cycle. On the scale of the entire memory system, this corresponds to coping an entire page into the secondary-port registers of the memory chips. Once this page transfer has been done, the parallel connections between the secondary-port registers and the bit-lines may be disabled, and their serial off-chip ports may be enabled. These now provide a second *totally independent, serial-access* memory port, which even has its own (asynchronous) clock. Through this "secondary serial-access port", the entire page can be transferred to the I/O device at the latter's own rate of transfer. This rate can be higher than the primary-port rate, because the shift-register is faster than the dynamic storage array. Of course, the inverse of the above scenario can be followed for transfers from the I/O device to the secondary-port register and finally into an arbitrary memory page.

The cost of this secondary memory-chip port is roughly similar to that of the sense-amplifiers in terms of silicon area: about 1/10th of the chip. In terms of pins, 3 additional ones are needed — one for the serial data, one for the serial clock, and one for controlling the mode of page transfer †. A hidden cost of this system arises from the fact that it requires the page-number to be the *row-address* instead of the column-address. The row-address is required to be available to the chip at the very beginning of the memory access. Thus, virtual-to-

† one additional serial data pin is required if simultaneous, synchronous input and output into/from the secondary-port register is desired.

physical address translation cannot be performed in parallel with row decoding and bit-line sensing, as it is possible when the page-number is used as column address.

In conclusion, the proposed secondary serial-access memory port will be advantageous for memories in systems with high I/O bandwidth, such that the CPU-I/O contention for memory cycles results in a heavier penalty than the address translation delay that the new memory organization may incur. In general, small additions or reorganizations of hardware may lead to changes in system architecture that can result in large gains in performance. In all cases, it is important to first identify the actual bottleneck areas so that the added hardware can be as effective as possible.

## Chapter 6.

## References.

- [ClGr77] D. Clark, C. Green: "An Empirical Study of List Structure in Lisp", Communications of the ACM, Vol.20, No.2, pp.78-87, Feb. 1977.
- [DEC81] Digital Equipment Co.: "VAX Hardware Handbook": System Throughput Considerations, Appendix K, p. 532 in 1980-81 edition.
- [DiML82] D. Ditzel, H. McLellan: "Register Allocation for Free: The C Machine Stack Cache", Proceedings, Symp. on Architectural Support for Progr. Lang. and Oper. Systems, Palo Alto, Ca, March 1982, (ACM: SIGARCH CAN vol. 10 no. 2, SIGPLAN Notices vol. 17 no. 4), pp. 48-58.
- [HaKe80] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames", CS292R-course final report, Univ. of California, Berkeley, June 1980.
- [Henn82] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill: "Hardware/Software Tradeoffs for Increased Performance", Proceedings, Symp. on Architectural Support for Programming Languages and Operating Systems, March 82, ACM SIGARCH-CAN-10.2 SIGPLAN-17.4, pp. 2-11.
- [Fuji83] R. Fujimoto: "VLSI Communication Components for Multicomputer Networks", Doctoral Dissertation, EECS, Univ. of Calif., Berkeley 94720.

August 1983.

- [PaSe81] D. Patterson, C. Séquin: "RISC I: A Reduced Instruction Set VLSI Computer," Proc. of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN 9.3, pp. 443-457, May 1981.
- [Patt83] D. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, K. VanDyke: "Architecture of a VLSI Instruction Cache", Proceedings, 10th Intl Symp. on Computer Architecture, ACM SIGARCH CAN, May 1983.
- [SeFu82] C. Séquin, R. Fujimoto: "X-Tree and Y-Components", Proceedings, Advanced Course on VLSI Architecture, Univ. of Bristol, England, ed. P. Treleaven, Prentice Hall, 1982.
- [Sher83] R. Sherburne: "Processor Design Tradeoffs in VLSI", Doctoral Dissertation, EECS, Univ. of Calif., Berkeley 94720, Dec. 1983.
- [Site79] Sites R. L.: "How to use 1000 registers", Proc. Caltech Conference on VLSI, Jan. 1979, pp. 527-532.
- [Smit82] A. Smith: "Cache Memories", ACM Computing Surveys, Vol.14, No.3, pp.473-530, Sept. 1982.
- [Thor84] J. Thornton: "Parallel operation in the Control Data 6600", AFIPS Proc. FJCC, pt.2 vol.26, pp.33-40. 1984. Also in: Bell, Newell: "Computer Structures: Readings and Examples", McGraw Hill Book Co., 1971, pp.489-496.

## CHAPTER 7:

## CONCLUSIONS.

Single-chip implementation of a general-purpose von Neumann processor offers advantages of low cost and of high performance owing to the high bandwidth of on-chip communications. However, even in Very Large Scale Integrated (VLSI) circuits, the limited transistors that are available on a single chip constitute a scarce resource when used to implement such a CPU. In this dissertation, the efficient use of these silicon resources was studied, showing that a Reduced Instruction Set Computer (RISC) architecture is advantageous, because it allows the integration of units providing fast access to operands and instructions, while still supporting the high-performance execution of the simple operations required during most part of general-purpose computations.

In chapter 2, the nature of general-purpose von Neumann computations was studied. Even programs that are heavily oriented to numerical floating-point computations execute an equally high number of array references, simple index arithmetic, and loop control instructions. In non-numerical programs, floating-point computations are replaced by copying or compare-and-branch instructions, and array references are often replaced by indirect accesses through pointers. This shows the crucial importance of fast operand accessing, simple address arithmetic, and fast compare-and-branch execution. The high

percentage of references to local scalar variables ( $\approx 60\%$  of all variable references) and the fact that most procedures have few such variables (a dozen or less) makes hardware support for fast operand accessing mandatory. The narrow range ( $\approx \pm 3$ ) of dynamic procedure nesting depth fluctuations for extended periods of time, makes a small set of overlapping register windows a viable approach.

The above findings led to the formulation of the Reduced Instruction Set Computer (RISC) concept and to the definition of the Berkeley RISC architecture, which became the basis of a large group project. Within this project, a second NMOS implementation of the Berkeley RISC processor, called RISC II, was designed, laid-out, debugged, and successfully tested after fabrication, in collaboration with Robert Sherburne.

The Berkeley RISC architecture, as described in chapter 3, is register-to-register oriented. It has simple instructions, a simple and orthogonal instruction-format, and a regular timing model that fits all instructions. Fast operand accesses are supported by a large register-file with multiple overlapping windows, where scalar arguments and local variables of procedures are allocated by default until the window is filled. Pipelining is utilized to keep the operational units busy executing the primitive operations selected by the compiler and optimizer. The combined effect of the 138 registers, organized in 8 windows, and of the 3-stage pipeline yields a machine of significantly higher performance than other commercial processors built in comparable technologies but with complex instructions and non-optimal use of registers. The size of programs compiled for RISC is not much larger than it is when compiled for other architectures, even though only simple instructions with a simple but code-inefficient format are available to the compiler.

The change away from the traditional trend towards instruction sets of increasing complexity resulted in a radical change in the allocation of the chip area to the various CPU functions. As shown in chapter 4, the control section of RISC II occupies only 10% of the area, in contrast with other processors with complex instruction sets, where the control and micro-program ROM occupy more than half the chip area. Scarce silicon resources are thus freed and used more effectively for the implementation of the large register file. The simplicity of the processor significantly contributes to its speed by reducing the number of gate-delays in the critical path and also by reducing the physical size, and thus the parasitic capacitances, of the circuit elements. Designing, laying-out, and debugging the simple RISC II processor required about five times less human effort than what is usual for other microprocessors. Chips were functionally correct and ran at the predicted speed on first silicon due to careful simulation and modeling, as described in chapter 5.

Future VLSI technology will allow the integration of larger systems on a single chip. Beyond multi-window register-files, such technologies will also allow on-chip support for fast access to other important elements of computations: instructions, and non-scalar operands. Chapter 6 proposed suitable organizations for such units. Remote-PC instruction-fetch-&-sequence units keep the program-counter and its associated logic close to the place where it is being used and provide reduced communication costs and more parallelism in executing jumps. When combined with a dual-port instruction-cache, they allow single-cycle compare-&-branch instructions and transparent unconditional branches, both of which are frequent instructions in general-purpose computations (about 1 out of 4 instructions). Instruction-caches also allow the CPU to effectively see two independent memory ports and make it possible to access data in memory while the next instruction is being fetched. Further inclusion of

a data-cache allows address-computation to partially overlap data access, thus making possible single-cycle data-memory-access instructions.

For the foreseeable future, Reduced Instruction Set Computer Architectures appear to be the most effective way to use the scarce chip resources to support the crucial need of general-purpose computations for fast access to operands and instructions.



## APPENDIX A:

DETAILED DESCRIPTION  
OF THE  
RISC II ARCHITECTURE

This appendix describes in detail the architecture of the RISC II CPU NMOS chip. It is a complement to chapter 3.

Some of the minor details of the RISC II architecture are "by-products" of its implementation. Most of these are points that were left unspecified in the original architecture. One important exception was mentioned in § 3.1.2. Register-indexed store instructions can only have an immediate source-2 -- not a register  $R_{s2}$  (fig. A.4.4). This restriction was imposed for implementation reasons. For similar reasons, this chip does not implement the pointer-to-register scheme (§ 3.2.3). On the other hand, it has some additional features: (1) conditional returns, and (2) compatibility with Expanding Instruction Caches [Patt83].

In this appendix the RISC II architecture is described from various points of view:

- definition of the user-visible CPU state (§ A.1);
- description of the CPU interface to the outside world (memory, I/O, interrupts) (§ A.2);

- discussion of the way in which instructions are sequenced (§ A.3);
- description of the instruction set, the instruction format, and the effect that the execution of each instruction has on the user-visible state of the CPU and on the CPU interface to the outside world (§ A.4);
- specification of the actions taken on interrupts/traps (§ A.5).

## A.1 User-Visible State of the CPU.

The User-Visible State (u-v state) of the CPU is the state of the processor chip that remains after execution of an instruction has completed, and before execution of the next instruction begins. The "future" of the processor depends, at that point, only on that state. The above "point in time", may not be well defined in the implementation, because of pipelining. However, the RISC architecture precisely defines the *state* of the CPU at that point, since it considers instruction execution to be indivisible and entirely sequential. The implementation must guarantee that the overall result of executing an arbitrary program on the real hardware will be the same as predicted by the architecture for purely sequential execution of the same program.

Here, we must draw a distinction between the "normal user" and the "interrupt-handler programmer" (i-h programmer) †. The "normal user" does *not* see the interrupts; what (s)he considers *one* instruction may in fact be an instruction that was aborted and then restarted after an interrupt handler completed its task. Such a user has his/her u-v state. The "interrupt-handler programmer" has a *finer-grain definition* of what an instruction is; for him/her interrupts and aborted instructions are clearly visible. Thus, (s)he needs to

---

† In this section the word interrupt is used to signify both interrupts and traps.

have additional visible state for the interrupt-handler to work with, while the normal user's visible state is left unaltered.

Figure A.1.1 shows the RISC II User-Visible State. The register file has 138 32-bit registers, accessible by an addressing pair: ( WindowNumber . Register-Number ). The WindowNumber ranges from 0 to 7 and is *always* provided by  $CWP = PSW \langle 12:10 \rangle$ . The RegisterNumber is provided by the individual instruction accessing the register; it ranges from 31 to 0, with 31 to 26 overlapping with the parent (caller), 25 to 18 being the locals, 15 to 10 overlapping with the child (callee), and 9 to 0 being the globals. Register-0 always has the (hard-wired) value 0; writing into it is allowed, but has no effect whatsoever.

The PSW has 13 bits containing various information. The CWP specifies the current window, and SWP is the limit-value for detecting register-file over/under-flows. The fact that they are shown to point to register-16 has no particular significance for the RISC II CPU; this is in analogy to the pointer-to-register scheme (§ 3.2.3). The I bit enables/disables interrupts. The S bit shows the current CPU privileges. The P bit is a place to save the previous value of the S bit when the latter is altered (set) on interrupts. The four remaining bits of PSW are the condition codes.

The SWP and the P bit are parts of the state visible by the i-h programmer only. Since register-file over/underflows are taken care of by an interrupt-handler, the "normal user" is given the illusion of a register-file with infinitely many windows ‡. The i-h programmer uses the SWP to detect over/underflows of the *real* register-file, the necessary window save/restore operations are carried out, and subsequently the value of the SWP is changed to reflect that fact. If the normal user reads SWP, (s)he may find "random values" in it (i.e. values that do

‡ If we wanted to be very precise, we should have shown that in fig. A.1.1, together with a CWP whose existence, but not its value, is known to the normal user.

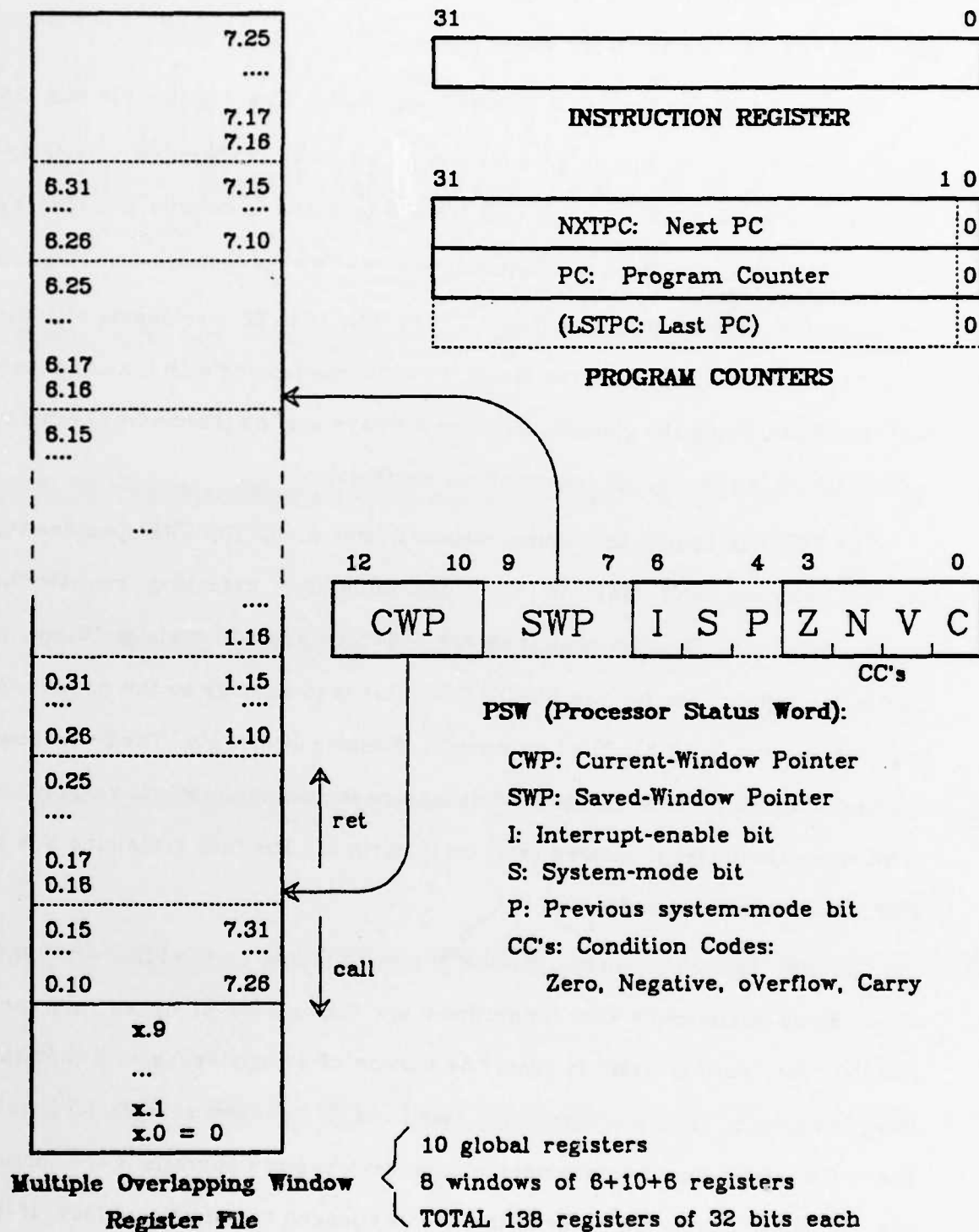


Figure A.1.1: User-Visible State of RISC II CPU.

not depend on his/her program alone). Similarly, the P bit is used to save a part of the normal-user-visible state before that latter is altered on an instruction abortion, so that it can be restored later; the normal user is given the illusion of an uninterrupted instruction execution.

The definition of the remaining four words of u-v state is more difficult and does not completely correspond to the implementation. The reason is that their values are of a *dynamic* nature. They are implicitly and automatically changed by the execution of *every* instruction; and thus cannot be *preserved* and read by some other instruction at a later time. However, they do belong to the u-v state, since they depend on the previous instruction(s), and they determine the future activities of the processor.

At the (conceptual) moment when an instruction completes execution and the next one has not yet started executing, the Instruction Register contains the instruction to be executed next. The PC contains the address of that next-to-be-executed instruction. This information would in general be redundant since the instruction is already known, but it is needed for PC-relative addressing.

At the end of an instruction the u-v state contains the address of the next instruction and also the instruction itself. This is a consequence of the delayed-branch scheme, which makes the instruction-fetch/execute pipeline visible to the user (§ 3.1.3). This also explains the existence of NXTPC in the u-v state. NXTPC is the address of the instruction to be executed subsequently after the one contained in the Instruction Register. It plays the role of PC in other computers. NXTPC is determined by the instruction that just finished executing, and not by the instruction that will execute next, in accordance with the delayed-jump scheme. The LSTPC register is part of the i-h programmer visible state only. It contains the address of the instruction which just finished executing. Its purpose is to hold the value of the PC when an instruction is aborted due to

an interrupt. The PC, in turn, holds the value of the NXTPC, while NXTPC is used for fetching the first instruction of the interrupt-handler. The three PC's always have a 0 least significant bit, since RISC II instructions are always half-word aligned in main-memory.

The u-v state remains unaltered during the "first part" of the execution cycle, when the processor reads some parts of it in order to compute some new values to be written back into the state during the "last part" of the execution cycle. In particular, PC-relative instructions will use the value that the PC has at the beginning of the execution cycle.

## A.2 Interface between CPU and Outside World.

The RISC II CPU communicates with the outside world by read and write accesses from/into a 4Gbyte virtual address-space. Thus, all I/O and System-Control functions are memory-mapped. Figure A.2.1 shows the signals that come in and out of the CPU.

For each memory access, the CPU issues an address and specifies the access type by the Read/Write signal. It also issues some additional information, which the peripheral devices may or may not use: (i) whether the processor is currently in user or system mode, and (ii) whether the CPU interprets the accessed item as instruction or as data (this signal is always "data" for write accesses). Addresses are virtual, 32 bit wide, and refer to bytes in memory.

The RISC II architecture understands and supports byte data (8 bit wide), half-word data (16 bit wide), and word data (32 bit wide). Figure A.2.2(a) shows the possible alignments that those types may have in memory and the address

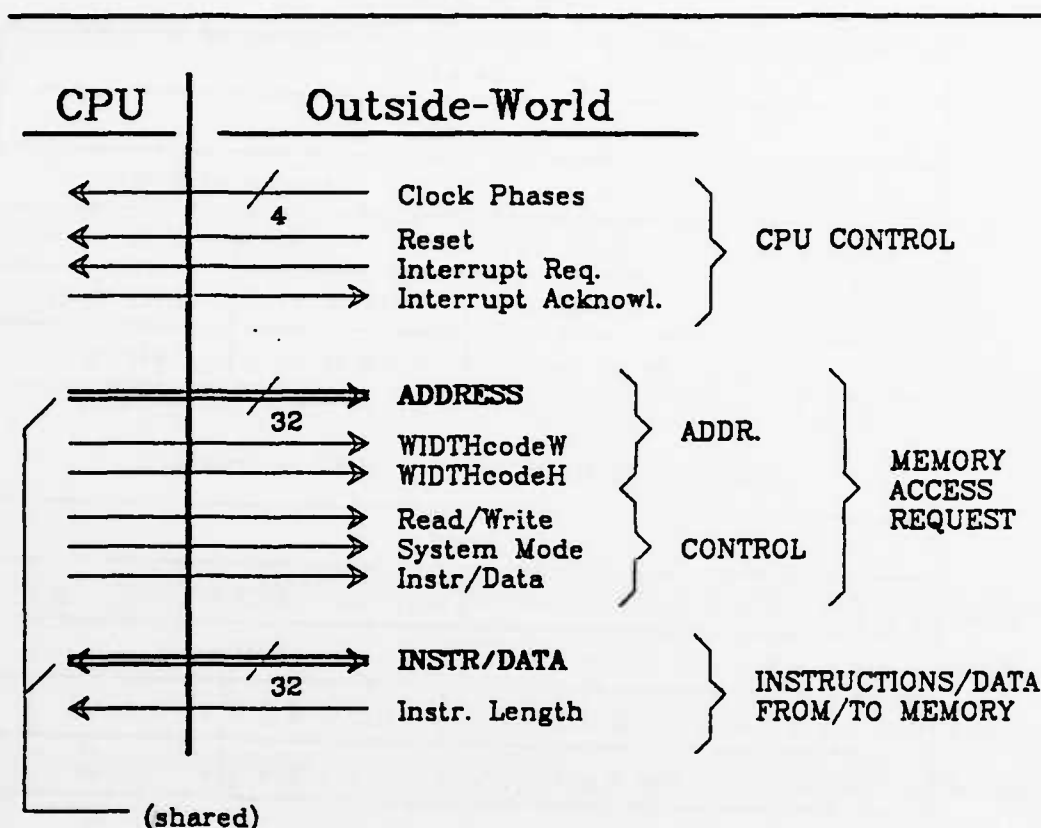


Figure A.2.1: CPU - Outside World Interface.

used to refer to each one of them (addresses shown are in decimal). Every item must be aligned so that, if the whole memory were packed with items of the same width and the same alignment, no item would cross word boundaries. The address of an item is the address of the least-significant byte in it, with addresses increasing towards more-significant bytes.

The memory of a RISC II system is organized in words, having, however, separate write-enable controls for the four byte-wide fields (byte-banks) in it, so that it can selectively write some but not all of the bytes in a word. Read accesses always read a whole word. The two least-significant address bits are discarded (considered to be zero) and the corresponding word is read and given to the CPU. Further selection and alignment of narrower data items is



## Data in MEMORY:

(a)

word at addr. 0				0
word at addr. 4				4
half-word @ 10		half-word @ 8		8
byte @ 15	byte @ 14	half-word @ 12		12
byte @ 19	byte @ 18	byte @ 17	byte @ 16	16
half-word @ 22		byte @ 21	byte @ 20	20

## Data in REGISTERS after a LOAD instruction:

(b)

	word																																
unsigned	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	s	half-word
signed	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	half-word
unsigned	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	s	byte
signed	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	byte

31	24	23	16	15	8	7	0
			19	18	13	12	

## Interpretation of IMMEDIATE fields of instructions (except ldhi):

(c)

s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	imm19
s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	imm13

## Destination REGISTER of a LDHI instruction:

(d)

imm19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

## Data in REGISTERS when operated upon:

(e)

32-bit quantity
-----------------

Figure A.2.2: Width, Addresses, and Alignment.

performed inside the CPU. For write accesses, the CPU always outputs a full 32-bit word onto the bus. However, only some of the bytes in that word are to be written into the corresponding bytes of the addressed word. Figure A.2.3 shows an example. The target word is determined as before by considering the 2 LS-bits of the address to be zero. The particular bytes to be written are determined by the two original least-significant address bits, and by the "width-code bits" (fig. A.2.1), according to the table in figure A.2.3. The width-code bits indicate the type of item to be written. They could be encoded in one bit, but the RISC II chip leaves them unencoded. Bit  $\langle i \rangle$  of the target memory word is written with bit  $\langle i \rangle$  of the word output on the bus (if the corresponding byte-bank is enabled); in other words, the memory does not need to perform any alignment.

Instruction fetches need not be different from data reads in a simple RISC II system. All instructions that the RISC II CPU understands are 32-bit wide. In fact, the RISC II CPU chip uses the same bus and timing for instruction and data fetches. However, the RISC II CPU is also compatible with Expanding Instruction Caches [Patt83]. These are instruction caches that receive from memory both 16- and 32-bit wide instructions. They "expand" the 16-bit ones into equivalent 32-bit instructions, which they subsequently supply to the CPU. All this is transparent for the CPU, which only sees 32-bit instructions, except for the fact that NXTPC has to follow the real memory address of the instructions, i.e. it has to get incremented by 2 or by 4, according to the length of the lastly fetched instruction. This is the purpose of the instruction-length signal going into the CPU (fig. A.2.1). In the absence of an expanding cache, this signal must be permanently set to "4 bytes". In the presence of such a cache, it is the cache who provides it.

Finally, the "outside world" has some control over the CPU by means of the clock phases and the reset- and interrupt-request signals. By means of the



clock phases, the CPU can be slowed down, or even stopped temporarily, which is useful for example, during the handling of a cache miss. A single interrupt request signal is provided. It is assumed that interrupt prioritizing is done outside the CPU. The reset signal acts as a non-maskable interrupt.

### A.3 Instruction Execution Sequencing.

RISC II sequences instruction execution in the usual von Neumann way, except for the visibility of the fetch/execute overlap which results in the delayed jump scheme (§ 3.1.3), and for the compatibility with Expanding Instruction-Caches (§ A.2). Figure A.4.5 describes control-transfer instructions.

The instruction-fetch process operates in parallel with the instruction-execute process and relatively independently from it. Out of the user-visible state, the fetch process uses the PC's, and the execute process uses the register file and PSW. The two processes communicate via the instruction register in the "forward direction" and via the control-transfer instructions in the "backwards direction". When an interrupt (or trap) occurs, progress of the execute process is aborted, and that process is inhibited from altering its u-v state. However, the fetch process is allowed to proceed, after an alteration is made to it so that it starts fetching instructions from the interrupt handler routine. These points will be discussed further in § A.5.

## A.4

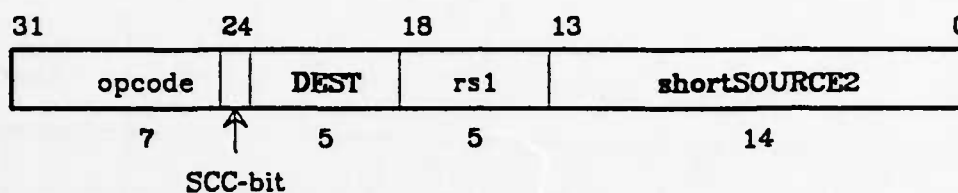
## Instruction Set.

Figure A.4.1 shows the two instruction formats of the RISC II CPU. Figure A.4.2 shows the binary and symbolic opcodes of the 39 RISC II instructions. The opcode uniquely determines whether the instruction has the short-immediate or the long-immediate format, and figure A.4.2 shows that correspondence. The opcode also uniquely determines the interpretation of the DEST field of the instruction (fig. A.4.1(a)), and fig. A.4.2 shows that correspondence as well. The format of the shortSOURCE2 field of the short-immediate instructions is determined by its leading bit, and not by the opcode (fig. A.4.1(b)). Figure A.4.2 also shows which instructions are privileged. An attempt to execute a privileged instruction while the System-mode bit is OFF causes an immediate trap which prevents the instruction from executing (§ A.5). A trap is also caused if execution of an illegal instruction (one with unassigned opcode) is attempted (notice that all opcodes with a leading bit equal to 1 are unassigned). RISC II has the same 39 instructions as RISC I, but it has different opcodes.

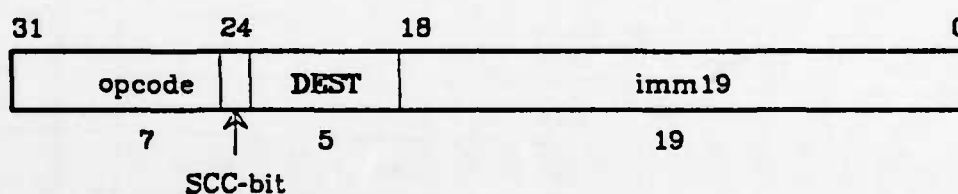
The instruction set can be subdivided into five groups of instructions, which are described in figures A.4.3 through A.4.8 and discussed in the next paragraphs.

**Register-to-register OP instructions:** Figure A.4.3 shows all the instructions of the second column of fig. A.4.2, except for *ldhi*. They include shift, logical, and integer-arithmetic operations. They all have the short-immediate format. They operate on register rs1 of the current window and on the source-2. Source-2 may be register rs2 of the current window or the immediate constant imm13 contained in the instruction. Registers are always interpreted as 32-bit quantities (fig. A.2.2(e)), and imm13 is always sign-extended (fig. A.2.2(c)). The result is written into register rd of the current window, and the instruction may

# 1. SHORT-IMMEDIATE FORMAT:



# 2. LONG-IMMEDIATE FORMAT:



# INSTRUCTION-FIELD FORMATS:

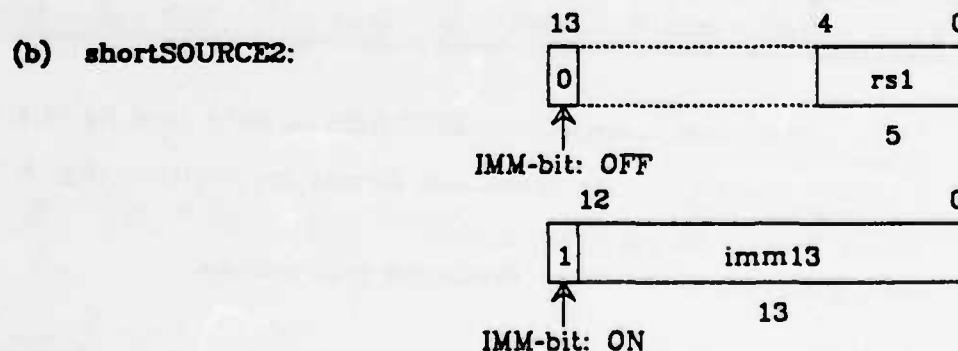
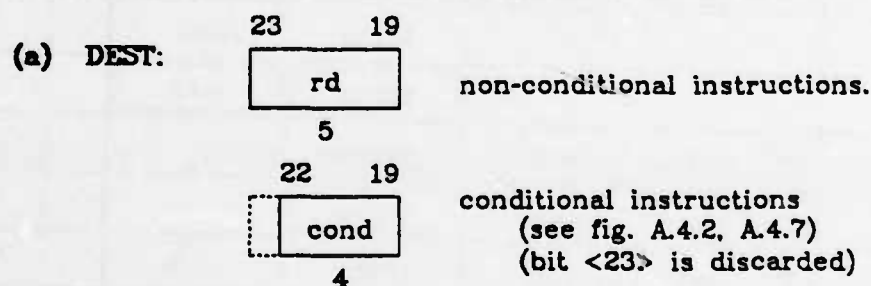


Figure A.4.1: RISC II Instruction Format.

	000xxxx	001xxxx	010xxxx	011xxxx	1xxxxxx			
xxx0000								
xxx0001	calli	sll						
xxx0010	getpsw	sra						
xxx0011	getlpc	srl						
xxx0100	putpsw	ldhi						
xxx0101		and						
xxx0110		or	ldxw	stxw				
xxx0111		xor	ldrw	strw				
xxx1000	callx	add	ldxhu					
xxx1001	callr	addc	ldrhu					
xxx1010			ldxhs	stxh				
xxx1011			ldrhs	strh				
xxx1100	(C) jmpx	sub	ldxbu					
xxx1101	(C) jmpr	subc	ldrbu					
xxx1110	(C) ret	subi	ldxbs	stxb				
xxx1111	(C) reti	subci	ldrbs	strb				

(C): conditional instructions: DEST-field is cond (see fig. A.4.1(a)).

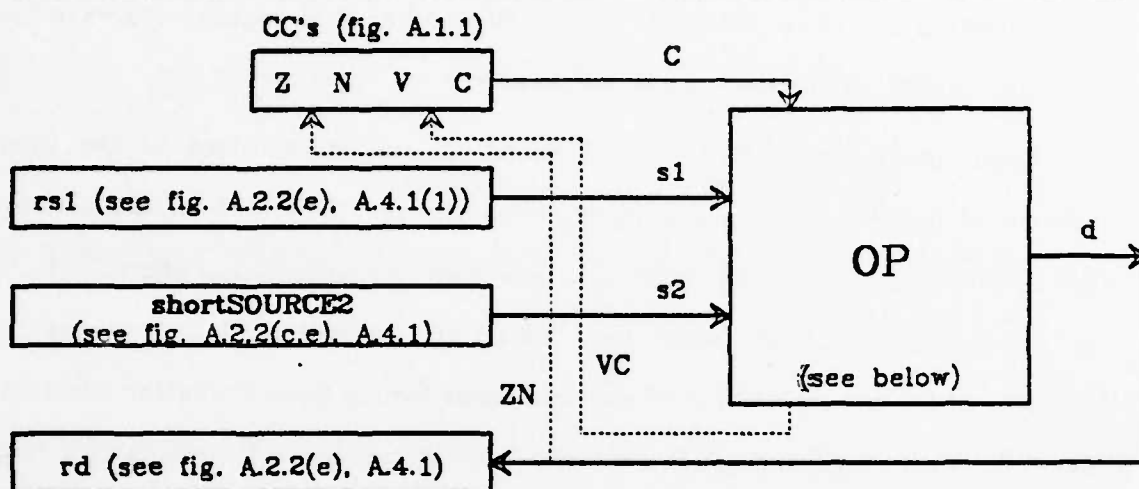
double boxes: long-immediate format instructions (fig. A.4.1(2)).

empty boxes: illegal opcodes.

calli, getlpc, putpsw, reti: privileged instructions.

Figure A.4.2: The RISC II opcodes.





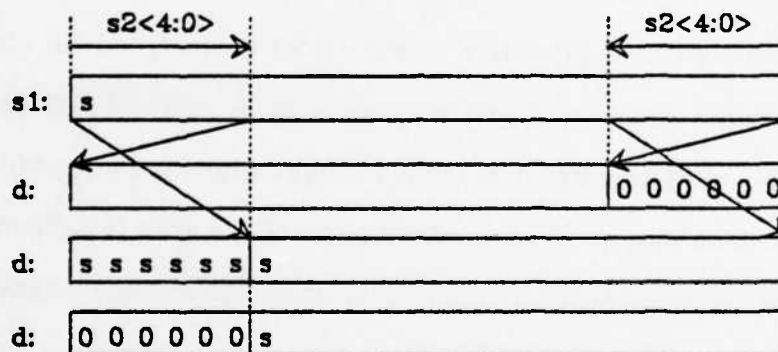
OP:

SHIFT:

sll:

sra:

srl:



LOGICAL:

(32-bit bitwise operations)

and, or, xor:  $d := s1 \text{ OP } s2$ ; (OP: AND, OR, or Exclusive-OR)

ARITHMETIC:

(32-bit 2's-complement operations)

add:  $d := s1 + s2$ ;

addc:  $d := s1 + s2 + C$ ;

sub:  $d := s1 - s2$ ; (internally:  $d := s1 + \text{NOT}[s2] + 1$ )

subc:  $d := s1 - s2 - \text{NOT}[C]$ ; (int.:  $d := s1 + \text{NOT}[s2] + C$ )

subi, subci:  $d := s2 - s1 \{-\text{NOT}[c]\}$ ;

CC'S: Updated iff the SCC-bit (instruction<24>) is ON, as follows:

$Z := [d=0]$ ;  $N := d<31>$ ;

shift, logical instructions:  $V:=0$ ;  $C:=0$ ;

arithmetic's:  $V := [32\text{-bit } 2\text{'s-complement overflow occurred}]$ ;

additions:  $C := \text{carry}<31>\text{to}<32>$  (assuming  $s1, s2$ : unsigned);

subtractions:  $C := \text{NOT}[\text{borrow}<31>\text{to}<32>]$  (for  $s1, s2$ : unsigned).

Figure A.4.3: ALU and Shift Instructions.

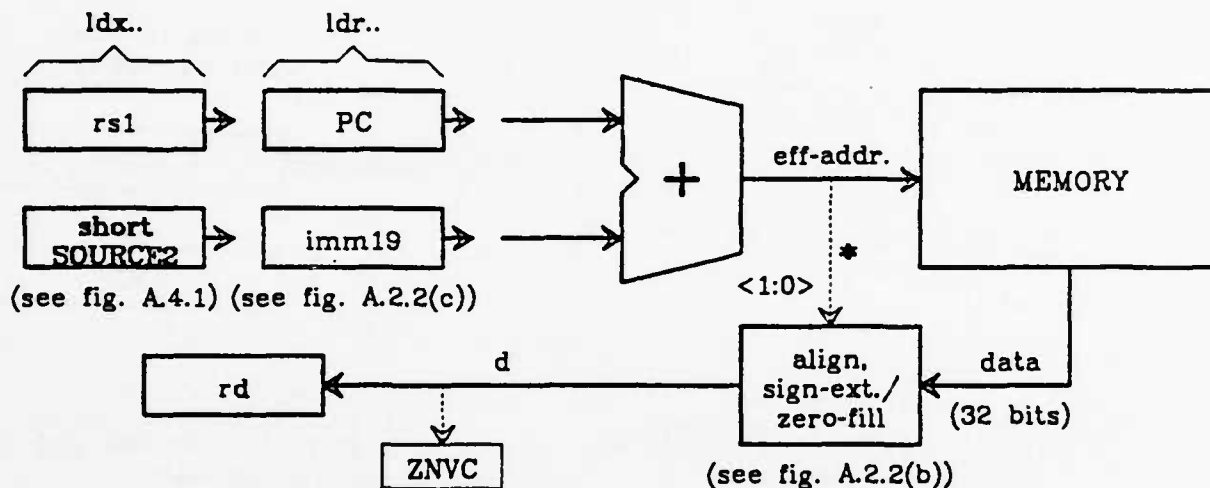
additionally update the Condition Codes. Remember that register-0 always has the value 0 and that writing into it has no effect.

**Load instructions:** Figure A.4.4 shows all the instructions of the third column of figure A.4.2. They perform read accesses from the virtual address space. The effective address for the access is the sum of *rs1* and *shortSOURCE2* for register-indexed loads (which have the letter *x* in their symbolic opcodes), or the sum of PC and *imm19* for PC-relative loads (which have the letter *r* in their symbolic opcodes). There are separate load instructions for words (*w*), half-words (*h*), and bytes (*b*). For the two latter cases, there are instructions for loading those quantities as signed (*s*) or unsigned (*u*) numbers. The addressed word is first read from memory, as discussed in § A.2. Then, the desired part of the word is extracted from it, right-aligned, sign-extended or zero-filled, and written into *rd*. (The "desired part" of the word is defined by the instruction and by the two least-significant bits of the effective address; see fig. A.2.2(b).) The CC's may optionally be updated according to the value placed into *rd*.

**Store instructions:** Figure A.4.4 also shows all the instructions of the fourth column of figure A.4.2. They perform write accesses into the virtual address space. The effective address for the access is computed in a fashion similar to that for loads, except that for register-indexed addressing, *shortSOURCE2* *must* be an immediate (§ 3.1.2). The CPU properly aligns the item to be stored, according to its width specified by the instruction type and to the 2 LS-bits of the effective address (fig. A.2.3). It also outputs information about the item's width, so that the memory selectively writes only some of the four byte-banks (§ A.2).

**Control-Transfer Instructions:** Figures A.4.5 through A.4.7 describe the jump, subroutine call, and return instructions. Figures A.4.5 through A.4.7 describe them. The (conditional) jump and (unconditional) call instructions

## LOAD INSTRUCTIONS:



Iff SCC-bit is ON:  
 $Z := [d == 0]$ ;  $N := d < 31$ ;  $V := 0$ ;  $C := 0$ .

\* TEST ALIGNMENT !!:  
 If bad (fig. A.2.2(a)):  
 ABORT INSTRUCTION,  
 TRAP to address:  
 80000000 Hexadec.

## STORE INSTRUCTIONS:

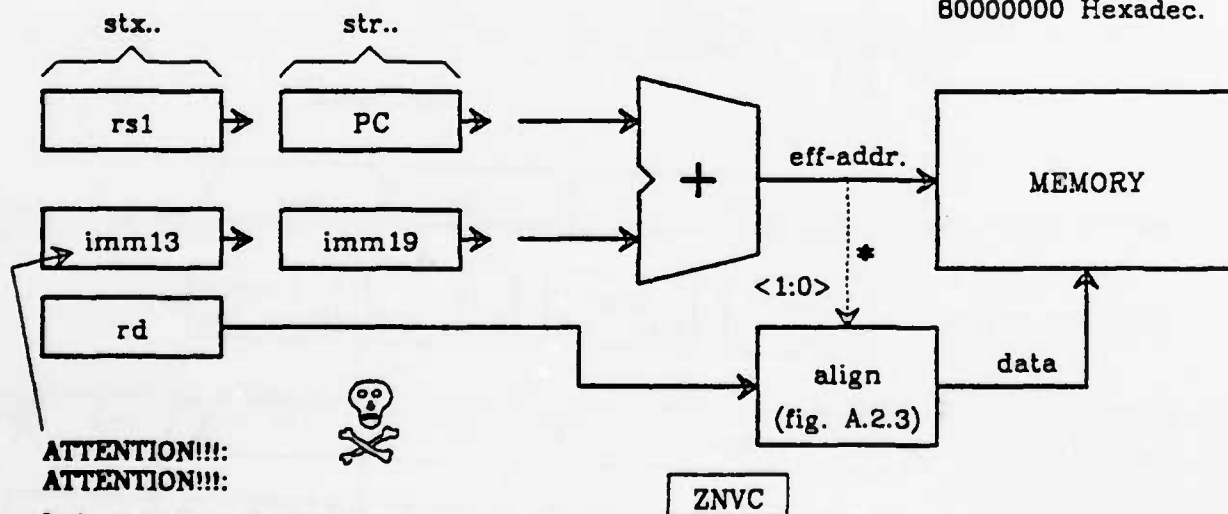
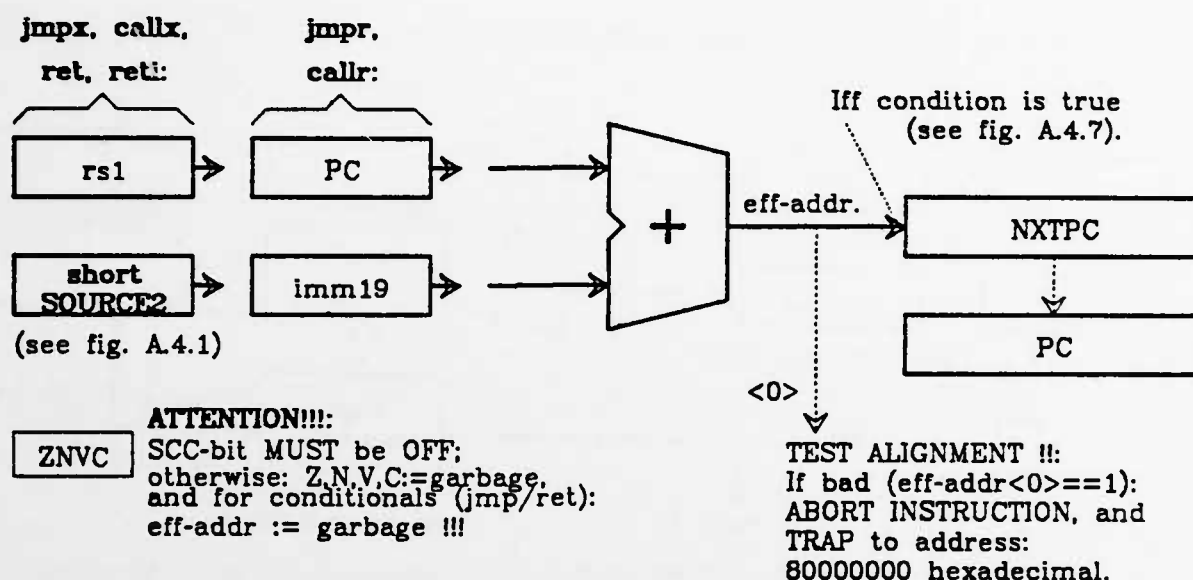


Figure A.4.4: Load and Store Instructions.



### DELAYED JUMP SCHEME:

(Result of Fetch/Execute Overlap)

Example:

100: ldrw ... PC+200;	204: sub ....
104: jmp, ... PC+100;	208: ...
108: add ....	300: data.
112: ....	....

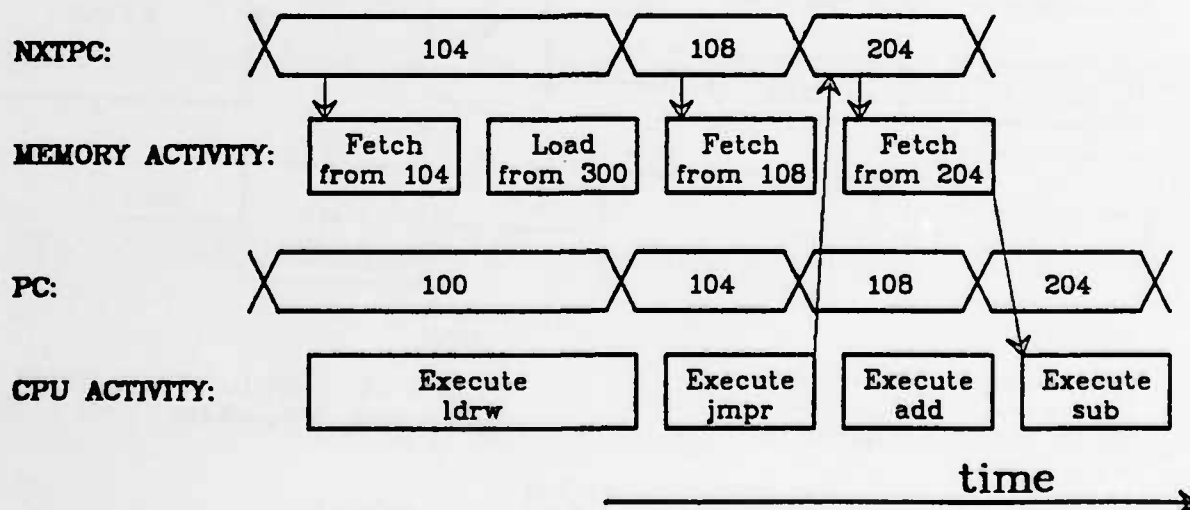


Figure A.4.5: Control Transfer - Delayed Jumps.

Figure A.4.6: Control-Transfer Instructions.

Instructions:	Effect & Notes:
<b>jmpx, jmpr:</b>	Iff condition is true (see fig.A.4.7), then control is transferred, as shown in fig. A.4.5.
<b>callx, callr:</b>	(1) Transfer Control (see fig. A.4.5); (2) $CWP := CWP - 1$ modulo 8 (change window - fig. A.1.1). (3) $rd := PC$ (save PC into destination-register); <b>NOTES:</b> (a) the $rs1$ (& $rs2$ ) register(s) specified in the instruction, are read from the OLD window; (b) the PC value that is saved is the PC of the call instruction itself; (c) the PC is saved into register number $rd$ of the NEW window; (d) if the change of CWP would result in a new value that would be equal to SWP (fig. A.1.1), then the call instruction is ABORTED, and the processor TRAPS to address 80000020 Hexadec. (if PSW_I is ON) (Reg-File Overflow occurred).
<b>ret:</b>	Iff condition is true (see fig. A.4.7), then: (1) Transfer Control (see fig. A.4.5); (2) $CWP := CWP + 1$ modulo 8 (change window - fig. A.1.1). <b>NOTES:</b> (a) the $rs1$ (& $rs2$ ) register(s) specified in the instruction, are read from the OLD window; (b) the normal use of this instruction is with target addr. $rs1 + 8$ (with $rs1 = rd$ of the call). (c) if the condition is true, and if the change of CWP would result in a new value that would be equal to SWP (fig. A.1.1), then the return instr. is ABORTED, and the processor TRAPS to address 80000030 Hexadec. (if PSW_I is ON) (Reg-File Underflow occurred).
<b>reti:</b>	Iff condition is true (see fig. A.4.7), then: (1) Transfer Control (see fig. A.4.5); (2) $CWP := CWP + 1$ modulo 8 (change window - fig. A.1.1). (3) Modify PSW: I:=ON (enable interrupts); S:=P . <b>NOTES:</b> Same as for ret.

compute their effective target address like the load instructions do (register-indexed or PC-relative). Only a register-indexed version of the return instruction is provided, because that's all that is needed. There is a return-from-interrupt instruction which will be discussed in § A.5. Return instructions are conditional. Figure A.4.6 contains the details of the window manipulation for call

Figure A.4.7: The RISC II Jump Conditions.

CODE	SYMBOL	NAME	MEANING
0001	gt	greater than (cmp signed)	$\overline{(N \oplus V) + Z}$
0010	le	less or equal (cmp signed)	$(N \oplus V) + Z$
0011	ge	greater or equal (cmp sign.)	$\overline{N \oplus V}$
0100	lt	less than (cmp signed)	$N \oplus V$
0101	hi	higher than (cmp unsigned)	$\overline{\overline{C} + Z}$
0110	los	lower or same (cmp unsign.)	$\overline{C} + Z$
0111	lo nc	lower than (cmp unsigned) no carry	$\overline{C}$
1000	his c	higher or same (cmp uns.) carry	$C$
1001	pl	plus (tst signed)	$\overline{N}$
1010	mi	minus (tst signed)	$N$
1011	ne	not equal	$\overline{Z}$
1100	eq	equal	$Z$
1101	nv	no overflow (signed arithm.)	$\overline{V}$
1110	v	overflow (signed arithmetic)	$V$
1111	alw	always	1

**CODE:** This is the "cond"-field (instruction<22:19>) (see fig. A.4.1(a)).

**SYMBOL:** This is how the condition is represented in Assembly.

**MEANING:** The condition is true if and only if  
the value of this function of PSW<3:0> is 1.

⊕: Exclusive-OR.

and return instructions (§ 3.2.2). Attention is drawn to the fact that the call instructions read their source register(s) from the old window, whereas they write their destination into the new one. Figure A.4.7 shows the branch

conditions employed by conditional-transfer instructions.

**Figure A.4.8: Miscellaneous Instructions.**

Instr.:	Effect & Notes:
<b>ldhi:</b>	(1) $rd := imm19 \ll 13$ — see figure A.2.2(d); (2) If SCC-bit (instr.<24>) is ON, then: $Z := [dest==0]$ ; $N := dest<31>$ ; $V,C := 0$ .
<b>getlpc:</b>	(1) $rd := LSTPC$ (fig. A.1.1); (2) If SCC-bit (instr.<24>) is ON, then: $Z := [LSTPC==0]$ ; $N := LSTPC<31>$ ; $V,C := \text{garbage}$ . <b>NOTES:</b> (a) the $rs1$ , $shortSOURCE2$ fields are discarded; (b) the value of $LSTPC$ , which is saved in $rd$ , is equal to the value that the PC had during the execution of the previous instruction. (c) this instr. is NOT transparent to interrupts.
<b>getpsw:</b>	(1) $rd := (-1)<31:13>$ concatenated $PSW<12:0>$ ; If SCC-bit is ON, (next-) CC's are set as by <b>ldhi</b> . <b>ATTENTION!:</b> (a) Previous instr. MUST have its SCC-bit OFF; (b) IMM-bit MUST be OFF, $rs2$ MUST be $r0$ (to prevent int-forw.). Otherwise $rd:=\text{garbage}!!$ <b>NOTES:</b> (a) see fig. A.1.1 for PSW; (b) $rs1$ is discarded.
<b>putpsw:</b>	(1) $PSW := [rs1 + shortSOURCE2]<12:0>$ . <b>ATTENTION!:</b> (a) the SCC-bit MUST be OFF; (b) the following instruction must NOT be: <b>callx</b> , <b>callr</b> , <b>calli</b> , <b>ret</b> , <b>reti</b> (i.e. NOT modify CWP), and must NOT set the CC's. (c) new PSW is NOT in effect during the first cycle following execution of this instr. <b>NOTES:</b> (a) see fig. A.1.1 for PSW; (b) $rd$ is discarded.
<b>calli</b>	(1) $CWP := CWP-1 \text{ modulo } 8$ (change window like <b>call</b> ). (2) $rd := LSTPC$ ; CC's possibly affected; like <b>getlpc</b> . <b>NOTES:</b> (a) $LSTPC$ is saved into $rd$ of the NEW window; (b) the $rs1$ , $shortSOURCE2$ fields are discarded; (c) if interrupts are enabled, an overflow trap may occur, like for <b>call</b> instructions. (d) this instr. is intended for use only by the H/W interrupt mech. (fig.A.5.1) — NOT by S/W.

**Miscellaneous instructions:** These are shown in figure A.4.8. The **ldhi** instruction is used for loading the most-significant part of long immediate constants into registers (fig. A.2.2(d)). The **getlpc** instruction is used as the first instruction of every interrupt-handler routine in order to save  $LSTPC$  into  $rd$



(see § A.5). The *getpsw* and *putpsw* instructions are used for manipulating the PSW in an arbitrary fashion, including saving it before interrupts nest, and restoring it at a later time. The *calli* instruction is used by the hardware interrupt mechanism. The *getlpc* and the *calli* read a part of the i-h programmer visible state, and their effect is therefore not transparent to interrupts. *Calli* is intended to be used only by the hardware interrupt mechanism, and *getlpc* should be used only as the first instruction of an interrupt handling routine (§ A.5) or at a place where interrupts are disabled. If these two instructions are executed in a different context, with interrupts enabled, their result depends on whether they themselves are interrupted or not. If they are not interrupted, they will yield the address of their previous instruction. If they are interrupted, they will yield the address of the last instruction of the interrupt-handler which serviced the interrupt.

## A.5 Interrupts and Traps.

Figure A.5.1 describes the automatic hardware actions which occur on interrupts and traps and the possible causes and respective priorities of interrupts/traps. The overall effect of an interrupt/trap is that it transfers control to the interrupt-handler, as if the interrupted instruction had never even started executing. (The only exception is an external interrupt not due to a page-fault during a memory-write cycle; see below.) Also, the interrupt/trap mechanism saves enough information, so that later the normal-user visible state can be reconstructed and the interrupted instruction can be restarted.

When an interrupt/trap occurs, the normal-user visible state (§ A.1) is protected from being altered by the executing instruction. The only exception is that a memory-write access in progress is allowed to complete if it can do so. This is possible if the interrupt was *not* due to a page-fault caused by that access. Furthermore, if execution has proceeded so far that write access has been started, no further trap (that is anomaly originating from the CPU) can occur during the execution of that instruction, since all traps occur during the address calculation cycle for load and store instructions. In other words, the only possible cause for the abortion of this instruction might have been an external *non-page-fault* interrupt. Therefore, if the memory-write access was allowed to complete, it must have been a *correct* access. When the same instruction is restarted later, after the interrupt-handling routine returns, it will repeat the same write-access, and the overall result will be the same as if the instruction had executed only once.

While most of the normal-user visible state is protected from being altered by the executing instruction, there are some parts of it that must be altered before the interrupt-handler can start executing. Those parts, then, must either be altered in a known and *reversible* manner, or their previous value must be saved before it is altered. When returning from the interrupt, the alterations must be reversed and the saved values restored.

The state which is altered in a reversible way is the PSW I bit and the CWP. The I bit was ON since the interrupt/trap occurred; and it is turned OFF. The CWP is decremented by the hardwired calli instruction. This decrementation will always occur, since the calli executes with interrupts disabled. Both of these changes are reversed by the reti instruction. Note that the I bit is not necessarily ON when a reset occurs; however, when a reset is used we don't care to save the previous state.

Figure A.5.1: Interrupts and Traps in RISC II.

Situation:	Activities, Effects, Notes:
Interrupt or Trap occurs:	<p><b>INTRODUCTORY NOTES:</b></p> <ol style="list-style-type: none"> <li>(1) Interrupts (i.e. external) and Traps (i.e. internal) are sampled/detected near the "middle" of every cycle;</li> <li>(2) Instructions "commit", i.e. modify the user-visible state of the CPU (fig. A.1.1), near the end of their "Execute" cycle.</li> </ol> <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> <p><b>AUTOMATIC (HARDWIRED) ACTIVITIES AND EFFECTS:</b>  Assume the Interrupt/Trap occurs during cycle#i.</p> <ol style="list-style-type: none"> <li>(1) The instruction executing during cycle#i is ABORTED, i.e. it is NOT allowed to "commit" -- Except that: (i) the PC's operate independently, and (ii) a memory-write that may have started will be allowed to complete (if it may).</li> <li>(2) The instruction that has been fetched during cycle#i (or i-1) is DISCARDED, and replaced by the (hardwired) instr.: <b>calli, SCC-OFF, rd=25, rsl-shortSOURCE2=garbage.</b></li> <li>(3) The PSW (fig. A.1.1) is modified as follows:  I:=OFF (disable interrupts); P:=S ; S:=ON (system-mode).</li> <li>(4) Instruction-Fetching starts at the address specified by the Interrupt-Vector, and NXTPC is loaded with that value.</li> </ol> </div> <p><b>INTERRUPT CAUSES, AND CORRESPONDING VECTORS:</b></p> <ol style="list-style-type: none"> <li>(1) Reset-pin pulsed high; Vector=80000000 Hexadecimal.</li> <li>(2) Interrupt-Request-Pin pulsed high; Vector=80000010 Hexad.</li> </ol> <p><b>TRAP CAUSES, AND CORRESPONDING VECTORS:</b></p> <ol style="list-style-type: none"> <li>(1) Illegal opcode (fig. A.4.2) executed; Vect=80000000 Hexad.</li> <li>(2) Privileged opcode (fig. A.4.2) executed while S==OFF (i.e. in user-mode); Vector=80000000 Hexadecimal.</li> <li>(3) Address-Misalignment (fig. A.4.4-5); Vect=80000000 Hexad.</li> <li>(4) Reg-File Overflow (fig. A.4.8(call)); Vect=80000020 Hexad.</li> <li>(5) Reg-File Underflow (fig. A.4.8(ret)); Vect=80000030 Hexad.</li> </ol> <p><b>INTERRUPT/TRAP DISABLING:</b>  All interrupts and traps, except the one caused by the Reset-pin, are disabled whenever the I bit of PSW is OFF.</p> <p><b>PRIORITIES:</b>  In case more than one interrupt/trap causes are present at once, the Vector is determined according to the priority: 80000000 has highest priority, 80000020 and 80000030 have medium priority (they cannot occur simultaneously), and 80000010 has lowest priority.</p> <p><b>NOTES:</b></p> <ol style="list-style-type: none"> <li>(1) In cycle#(i+1), the hardwired calli instruction will execute, changing the window, and saving LSTPC into reg. 25 of the new window. The value saved is equal to the PC of cycle#i.</li> <li>(2) In cycle#(i+1), the instr. <math>\odot</math> Vector will be fetched, and it will execute in cycle#(i+2). That instruction must be a getlpc, to save LSTPC of cycle#(i+2) = NXTPC of cycle#i.</li> </ol>

The part of the state that is saved before it is altered consists of the PSW S bit, the NXTPC, and the PC. The S bit is saved into the P bit by automatic hardwired action, and it is restored by the *reti* instruction. The PC is saved into LSTPC, which is subsequently saved into register 25 of the new window by the *calli* instruction. The NXTPC goes into PC, which then goes into LSTPC, and which must be saved by the first instruction of the interrupt-handler. That instruction *must* be a *getlpc*, and it will place this value typically into register 24. After the completion of the interrupt-handling, these two values must be restored in the following fashion:

```
jmpx (alw), r25 + 0 ;
```

```
reti (alw), r24 + 0 .
```

Notice that the detection of register-window overflows works in such a way that registers 25 through 16 (the locals) of the window just below the current one, are guaranteed to be free whenever interrupts are enabled, provided this condition held true when interrupts were originally enabled (§ 3.2.2). The *calli* and *getlpc* instructions save LSTPC into two of those local registers on interrupts. The interrupt-handler may also use those local registers (and only those) as scratch memory.

While the normal-user visible state is saved on an interrupt/trap, the i-h programmer visible state (§ A.1) is *not* saved. This means that interrupts/traps must NOT nest without prior appropriate arrangements. Thus, before the interrupt-handler re-enables interrupts, or modifies the CC's, or uses any registers other than 25 through 16, or calls a subroutine, it *must* save PSW somewhere (e.g. *getpsw r23, r0, r0*), and make sure that there are more free windows below the current one. As the last step before returning from the interrupt-handler with the instruction-pair *jmpx-reti*, the PSW must be restored (e.g. *putpsw r0, r23, r0*). This restores the environment and the state that existed

before this interrupt occurred.

## Appendix A.

## References.

- [Patt83] D. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, K. VanDyke: "Architecture of a VLSI Instruction Cache", Proc. of the 10th Symposium on Computer Architecture, ACM SIGARCH CAN 11.3, pp. 108-116, June 1983.

Processor Design Tradeoffs in VLSI

By

Robert Warren Sherburne, Jr.

B.S. (Worcester Polytechnic Institute) 1978

M.S. (University of California) 1981

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:

*Carlo H. Séguin* 4/13/1984  
Chairman Date  
*James I. Smith* 4/16/1984  
*Daniel A. Holger* 4/6/84

## PROCESSOR DESIGN TRADEOFFS IN VLSI

*Robert Warren Sherburne, Jr.*

### ABSTRACT

As the density of circuit integration is increased, management of complexity becomes a critical issue in chip design. Hundreds of man-years of design time are required for complex processors which are presently available on a few chips. This high cost of manpower and other resources is not acceptable. In order to address this problem, the Reduced Instruction Set Computer (RISC) architecture relies on a small set of simple instructions which execute in a regular manner. This allows a powerful processor to be implemented on a single chip at a cost of only a few man years. A critical factor behind the success of the RISC II microprocessor is the careful optimization which was performed during its design. Allocation of the limited chip area and power resources must be carefully performed to ensure that all processor instructions operate at the fastest possible speed. A fast implementation alone, however, is not sufficient; the designer must also consider overall performance for typical applications in order to ensure best results. Areas of processor design which are analyzed in this work include System Pipelining, Local Memory Tradeoffs, Datapath Timing, and ALU Design Tradeoffs. Pipelining improves performance by increasing the utilization of the datapath resources. This gain is diminished, however, by data and instruction dependencies which require extra cycles of delay during instruction execution. Also, the larger register file bitcells which are needed in order to support concurrency in the datapath incur greater delays and reduce system bandwidth from the expected value. Increased local memory (or register file)



capacity significantly reduces data I/O traffic by keeping needed data frequently in registers on the chip. Too much local memory, though, can actually reduce system throughput by increasing the datapath cycle time. Various ALU organizations are available to the designer; here several approaches are investigated as to their suitability for VLSI. Carry delay as well as power, area, and regularity issues are examined for ripple, carry-select, and parallel adder designs. First, a traditional, fixed-gate delay analysis of carry computation is performed over a range of adder sizes. Next, delays are measured for NMOS implementations utilizing dynamic logic and bootstrapping techniques. The results differ widely: the fixed-delay model shows the parallel design to be superior for adders of 16 bits and up, while the NMOS analysis showed it to be outperformed by the carry-select design through 128 bits. Such a result underscores the need to reevaluate design strategies which were traditionally chosen for TTL-based implementations. Single-chip VLSI implementations impose a whole new set of constraints. It is hoped that this work will bring out the significance of evaluating the design tradeoffs over the whole spectrum ranging from the selection of a processor architecture down to the choice of the carry circuitry in the ALU.

In this research I was supported for three years by a General Electric doctoral fellowship. The RISC project was supported in part by ARPA Order No. 3803 and monitored by NESC #N00039-78-G-0013-0004.

## Table of Contents

<b>Chapter 1:</b>	
<b>INTRODUCTION .....</b>	<b>1</b>
<b>Chapter 2:</b>	
<b>SYSTEM PIPELINING .....</b>	<b>7</b>
<b>Chapter 3:</b>	
<b>LOCAL MEMORY TRADEOFFS .....</b>	<b>19</b>
<b>Chapter 4:</b>	
<b>DATAPATH TIMING .....</b>	<b>34</b>
<b>Chapter 5:</b>	
<b>ALU DESIGN TRADEOFFS .....</b>	<b>45</b>
<b>Chapter 6:</b>	
<b>PROCESSOR PERFORMANCE .....</b>	<b>63</b>
<b>Chapter 7:</b>	
<b>CONCLUSIONS .....</b>	<b>78</b>

# CHAPTER 1:

## INTRODUCTION

In the world of integrated circuits a revolution is taking place. Silicon chips, which only a decade ago contained several transistors or logic gates, now accommodate up to hundreds of thousands of transistors. Several 32-bit Central Processing Unit (CPU) implementations, as well as 64 and 256 Kilobit dynamic Random Access Memories (RAMs), have been produced on a single chip. Higher levels of integration offer systems which are not only smaller, cheaper, and less costly to operate than their predecessors: they offer higher performance as well. By shrinking the circuitry so that it can reside on less than a square centimeter of silicon area, wire delays are reduced dramatically. As a result, the user obtains higher performance at lower cost.

The rising complexity confronting the designer is a serious concern as device capacity on a chip increases. The design of a typical, 32-bit microprocessor requires 30 or more man-years. If this were performed by a single person, the fabrication technology will have changed so much that the original assumptions made regarding chip constraints would be grossly invalid. In order to shorten the elapsed design time, chips are partitioned into modules, each of which is constructed by a design team. Each team optimizes its module while conform-

ing to the specifications assigned by the project leader or manager. This divide-and-conquer approach is the traditional methodology in industry, where early product release yields a high return on the initial investment.

A disadvantage of the divide-and-conquer strategy is that no design team is familiar with the chip as a whole. This makes global optimization difficult, if not impossible. The overall organization of the system and its microarchitecture sets a fundamental limit on performance. A poor choice of microarchitecture will render a design doomed to a short life, if not outright failure, in the marketplace. The microarchitect's responsibility is to address this issue. He must be familiar with the architecture as well as the constraints of the fabrication technology. Since the chip constraints are constantly changing, design decisions must be periodically reevaluated.

Traditional CPU's such as in the IBM 360/370, DEC VAX-11/780, and so forth consist of several circuit boards, filled with standard Small and Medium-Scale Integration (SSI and MSI) packages. Performance is limited by the logic delays and wiring delays associated with the many inter-chip communication paths. Bipolar chips are normally used in such designs because they offer high transconductance. This means that a greater output current drive is available, which is important for reducing wire delays. Increasing the speed of signal propagation between chips requires more power per chip. Expensive cooling systems must then be added in order to control chip temperature.

In contrast, more recent 32-bit CPU designs have been implemented by Very Large-Scale Integration (VLSI) on a single chip. As a larger fraction of the system is placed on a chip, interchip wiring delays are reduced. Interchip delays are replaced with smaller, on-chip wire delays, increasing system performance. A ceiling for maximum transistor count is set by the limited chip area and power dissipation of the technology. Because Metal-Oxide Semiconductor

(MOS) transistors are smaller and consume less power than their bipolar counterparts, they are more attractive for VLSI. The poor transconductance of the MOS transistor increases off-chip signal delay, but this is offset by the reduced number of such delays inherent in higher levels of integration. MOS is presently the most popular fabrication technology for VLSI systems.

A VLSI processor is expensive to develop. Instead of relying on available SSI/MSI parts, the designer must perform circuit design, layout, and simulation at the device level. Optimization of one module on the chip affects the area, power, and timing available for the other modules. Wiring is costly in terms of chip resources: a 32-bit bus occupies a large amount of area in the planar layout. The number of Input/Output (I/O) pads is limited by chip periphery. This complicates testing by restricting access to internal state. Redesign is costly because new masks and wafers are required, delaying the product several months. As a consequence, the board-level design is more attractive for implementing complicated processors.

The high costs associated with a single-chip design are mainly due to complexity in design and testing. These penalties of single-chip implementation, can be alleviated by simplifying the CPU design. By reducing complexity and internal state of the machine, it will be simpler to design and test, and will be more likely to be free of design errors on the first try.

Several NMOS, single-chip VLSI microprocessors have been designed with this idea in mind. The RISC I [1], RISC II [2], and MIPS [3] implementations utilize low-level instructions, each of which requires a single machine cycle to execute. This regular execution timing simplifies the application of pipelining for high performance while remaining conceptually simple. Less instruction decoding is necessary, allowing small, fast Programmed Logic Arrays (PLAs) or simple decoders to be used.

This contrasts with commercial single-chip microprocessors, which dedicate over half the die area to microcode Read-Only Memory (ROM) for instruction decoding. In the Reduced Instruction Set Computer (RISC) design, area freed up by the reduced control circuitry is used for an expanded register file. Depending on the application environment, other functions may be incorporated on chip instead with the freed-up area to improve system performance.

In a single-chip design, datapath speed is a limiting factor in system performance. The datapath consists of the functional modules which manipulate data and provide for its temporary storage. Additionally, as its name implies, it includes the paths over which data may flow between these modules. Datapath cycle time is determined by delays in the main datapath modules and the communication overhead incurred to and from these modules during the machine cycle.

In order to minimize the datapath cycle time, the microarchitect determines what datapath modules are necessary and how they are orchestrated during the cycle. First, a functionally partitioned, two-dimensional representation of data flow is composed. Timing schemes are formulated which maximize concurrency of data operations and data flow. Next, the modules are optimally placed on the chip for minimum communication path delay and area. Several iterations may be required in order to stay within the limits of the chip resources.

This thesis studies fundamental design tradeoffs of importance to the microarchitect of a VLSI processor. All of these tradeoffs are interrelated; the microarchitect must decide which tradeoffs to make for best performance under specified conditions. These conditions include the fabrication technology and amount of chip resources available, as well as the type of environment for which the system is designed for.

Pipelining at the system level is investigated in Chapter 2. It presents the timing of the basic operations to be performed using increasing levels of concurrency. It is when overall datapath timing and resource allocation are determined that optimal topology of information flow must be considered.

Within the datapath itself, the local memory (register file) and ALU delays limit the speed which may be attained. A larger local memory effectively reduces data traffic arising from procedure calls and returns. Datapath bandwidth, however, is reduced due to the increased register cycle time. A conflict then exists between the desire for maximum datapath bandwidth, and the need to reduce data I/O overhead.

Since the local memory may occupy a significant portion of the chip, it is important to ensure that it makes effective use of available resources. Programming environments which include many nested procedures benefit significantly from a multiple-bank local memory scheme. On the other hand, those with few procedures may suffer from the increased register cycle time. In addition to the programming environment, the register-bank swapping strategy, overflow interrupt overhead, and data I/O bandwidth affect optimal memory size. These tradeoffs are investigated in Chapter 3.

Datapath bandwidth may be improved by pipelining the read and write operations of the register file. Different bit cells are required for different levels of pipelining. In general, the number of wordlines and bitlines in the cell must increase with higher concurrency. Register cell design issues relating to datapath timing are investigated in Chapter 4.

ALU delay can also be improved with increased parallelism. Adder delay analysis has traditionally been performed using the notion of fixed gate delay. This is appropriate for TTL implementations where performance is dominated by on-chip buffer delay. Such an approach is not suitable for VLSI, where



performance becomes limited by wiring and transistor parasitics. In Chapter 5, relative performance of ripple, partial lookahead, conditional carry, and parallel adders is analyzed and compared for both the constant gate delay model and an NMOS model which takes into account device parasitics and permits evaluation of alternative circuit design strategies for increased performance.

Interactions between these areas of design tradeoffs are investigated in Chapter 6. Designing for limited chip area and power resources is also discussed. Conclusions drawn from these areas of analysis are summarized in Chapter 7.

### References

- [1] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer," Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.
- [2] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture," Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.
- [3] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.

## CHAPTER 2:

# SYSTEM PIPELINING

The goal of pipelining is to make more effective use of system resources, and in so doing, increase performance. Greater levels of pipelining lead to increased concurrency. The execution of one instruction (or microinstruction) may overlap initiation and completion of several others. This will be illustrated for datapath timing in a later chapter. In this chapter we will take into account the interaction with external (off-chip) memory.

### Performance Improvement by Pipelining

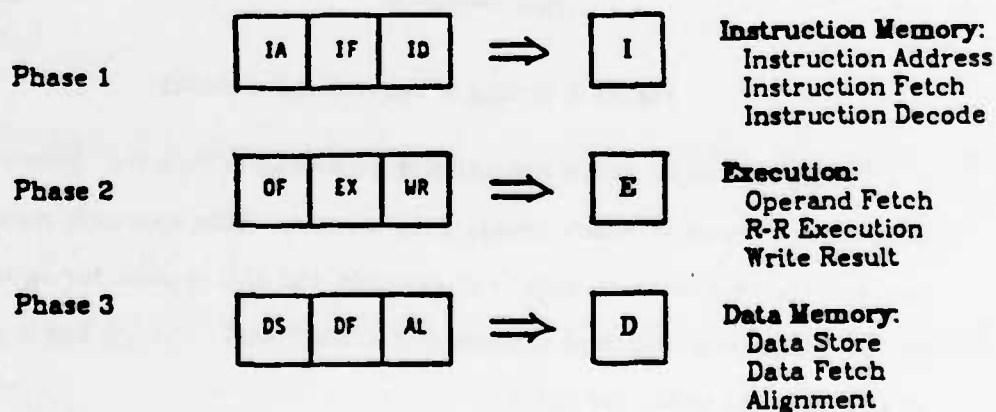


Figure 1: The Three Phases in Instruction Processing

There are several sequential steps involved in the execution of a single memory-to-memory instruction. The first phase of this cycle consists of the instruction fetch and decode. The next phase encompasses the register-to-register operation within the CPU, where data modifications are performed. For data I/O (Input/Output) instructions, this cycle consists of address calculations for LOADs, STOREs, or JUMP's. Finally, there is a third phase for LOAD and STORE instructions during which they access data memory. Support for sub-word data (e.g. bytes or half-words) may be included here. These three phases, each subdivided into three subphases, utilize different resource groups (Figure 1).

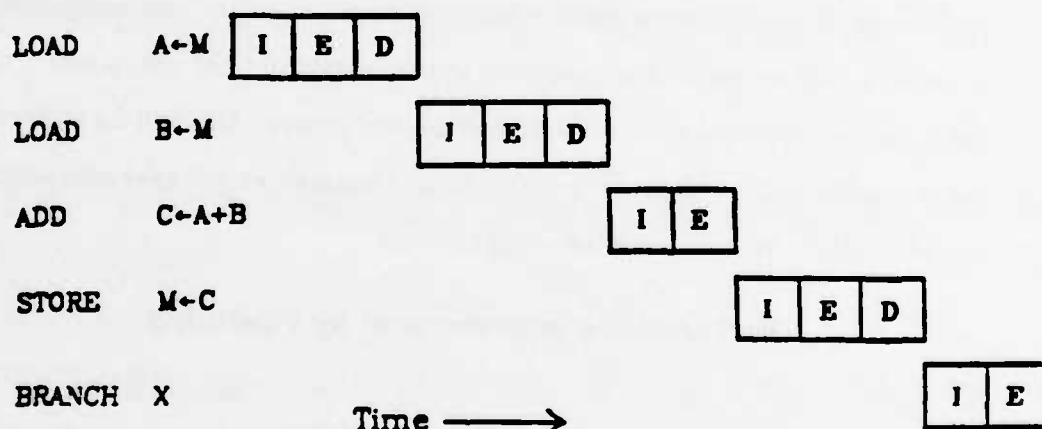


Figure 2: Timing of Sequential Execution

Timing for simple, serial execution is illustrated in Figure 2. Shown are five instructions, three of which access data memory. This approach makes poor usage of the resources on chip. For example, the ALU is used during less than half of the phases (one third for data I/O instructions). The I/O bus is not used at all during the execution phase.

Even very simple pipelining can increase performance substantially. A 2-way pipelined scheme is shown in Figure 3, in which instruction fetching overlaps execution of the previous instruction. This yields up to twice the bandwidth

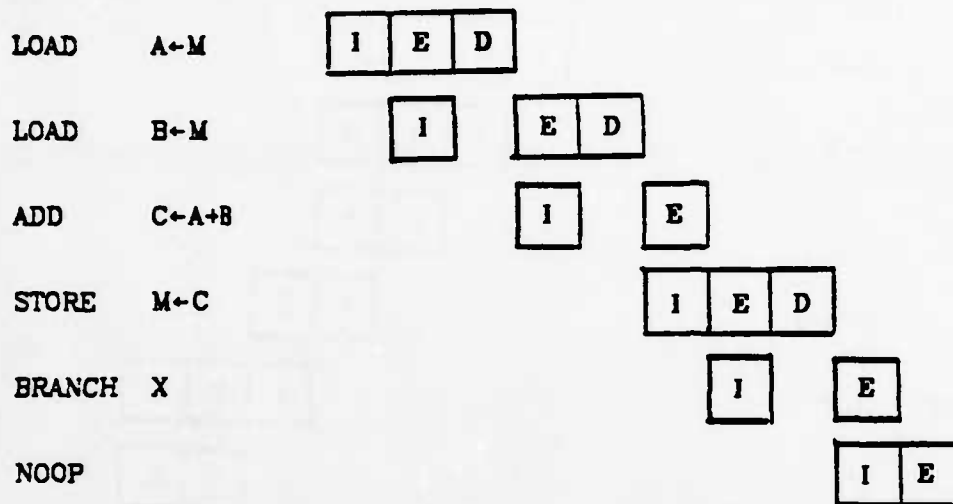


Figure 3: Two-Way Pipelined Timing Showing Branch Delay

of the serial scheme. It is assumed that only a single I/O operation is permitted in each phase, thus a wait-state is inserted while data I/O takes place. Hence, performance is I/O limited – a single memory access occurs in each phase. This is the timing scheme of the RISC I and RISC II microprocessors [1].

In the event of a program branch, the branch address calculation is not completed until the following instruction has been fetched. This results in a delay of one phase before the target address is ready. In order to accommodate this delay, a NOOP (NO OPERATION) instruction is inserted after the branch instruction. This creates overhead for all program branches. This overhead may be reduced by redefining the branch instruction in such a way that it is supposed to take effect only after the subsequent instruction. The code can then be reordered so that the instruction after the branch performs useful work prior to the branch occurrence [2]. Otherwise, a wait-state (in the form of a NOOP) is required. Code reorganization for this "delayed branch" optimization will be discussed in more detail later in this chapter.

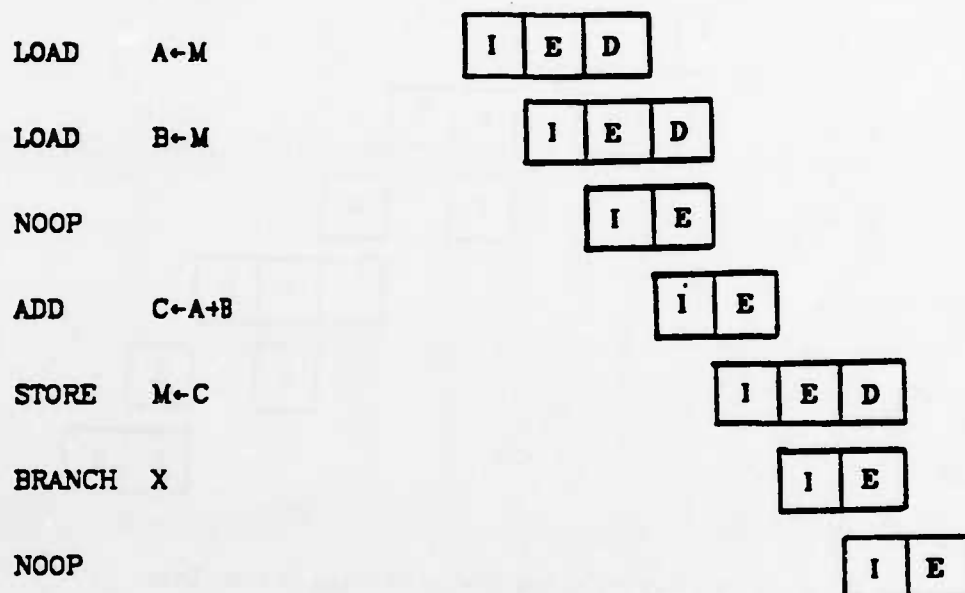


Figure 4: Three-Way Pipelined Timing Showing Data Delay

The pipelining can be further improved by permitting two memory I/O operations per phase, as shown in Figure 4. Multiple I/O operations per phase may be attained either by multiplexing a single port or by replicating ports. In this pipelining scheme a new instruction is initiated each phase, and as many as three instructions may overlap. Unoptimized branch delay remains at one phase, as in the previous scheme. However, the overlapped data memory access may now require wait states following LOAD instructions. As indicated in the figure, the data fetch is not completed prior to the execution of the following instruction. If this following instruction utilizes the fetched data as one of its operands, it must wait one phase. Code reorganization for data dependency optimization is similar to that for delayed branches (to be discussed). At best, this approach is up to three times faster than that of the sequential approach.

A four-way pipelined timing scheme is detailed in Figure 5. The execution phase is divided into two sub-phases:  $E_1$  (register file read, with internal

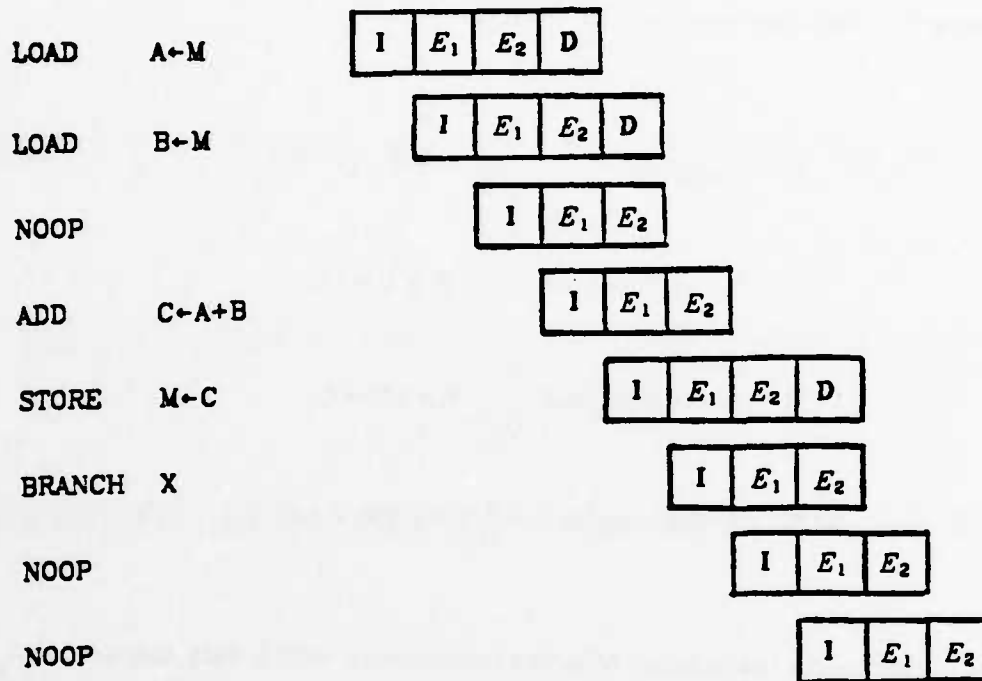


Figure 5: Four-Way Pipelined Timing

forwarding and overlapped write) and  $E_2$  (ALU or shifter operation). The ALU and register file are used in every phase; this allows maximal resource usage. Overhead due to unoptimized data dependencies remains a single phase if the ALU result is directly forwarded to the concurrent read phase of the following instruction. Unoptimized branch overhead, however, is now two phases per branch. Two memory accesses per phase must be supported. This requires two I/O busses, as in the TMS 320 [3] and the MIPS [4] microprocessors. Additionally, the memory I/O cycle is now shortened to match half the execution time, so faster memory is necessary in order to realize the full gain of this added level of pipelining.

The processor-limited execution time for a given program using the various pipelining schemes discussed may be given as:

- |      |                 |                                       |
|------|-----------------|---------------------------------------|
| I.   | Sequential      | $2N + D$                              |
| II.  | 2-Way Pipelined | $N + D + (J)$                         |
| III. | 3-Way Pipelined | $N + (D) + (J)$                       |
| IV.  | 4-Way Pipelined | $\frac{1}{\alpha} [ N + (D) + (2J) ]$ |

where  $N$  represents the number of coded instructions, with  $D$  data accesses and  $J$  jumps. Time is normalized with respect to a single register cycle. Optimizable overhead (branching, data dependencies) is shown in parentheses. Ideally, the factor  $\alpha$  is 2 for the 4-way scheme, assuming that the factor  $\alpha$  represents the available memory cycle speedup available over the previous schemes. Maximum performance improvement occurs for  $\alpha=2$ .

A comparison of ideal performance (fully optimizable) for these approaches is shown in Figure 6. Results are normalized with respect to the non-pipelined (sequential) case; the four-way scheme assumes  $\alpha=2$ . Bandwidth reduction caused by added data I/O cycles is shown in the shaded areas on the graph; the full shaded penalty occurs for the case of all instructions performing data LOADs. A fraction of this area would then pertain to the actual data I/O cost for a particular program.



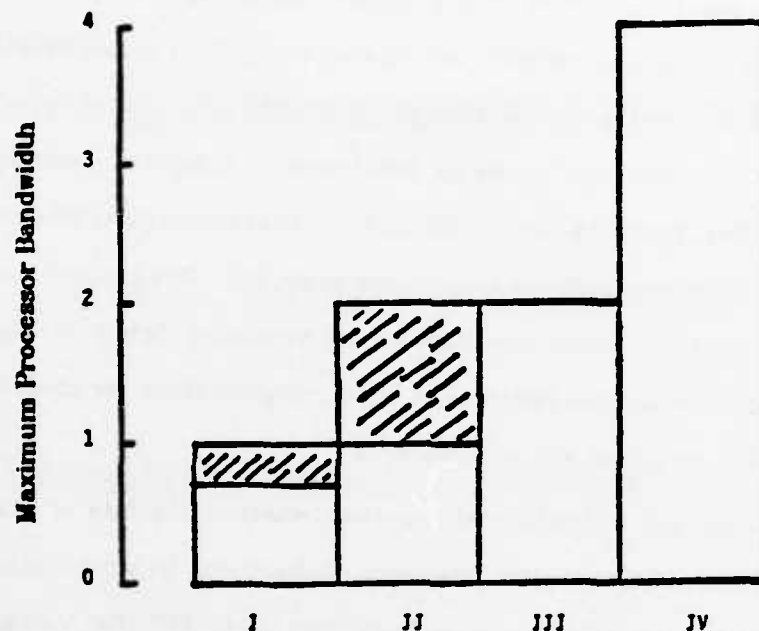


Figure 6: Performance Comparison of Pipelined Schemes (for  $\alpha=2$ )

(shaded area indicates maximum possible data I/O overhead)

### Optimization of Pipeline Dependencies

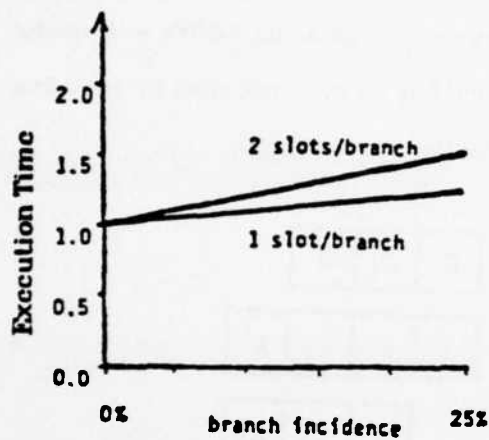
Ideal performance is proportional to the number of pipelining levels employed in the system. However, overhead due to data and instruction dependency reduces the efficiency of pipelining. This overhead is most significant for highly-pipelined machines. Code reorganization at the register-to-register instruction level may reduce this penalty inherent in pipelined implementations. The effective use of on-chip local memory also reduces data dependencies, by reducing data I/O traffic. Design tradeoffs concerning local memory will be investigated in a later chapter.

Code is optimized by reordering so that the required data or instruction is available when needed. The optimized jump or load (in the event of a branch

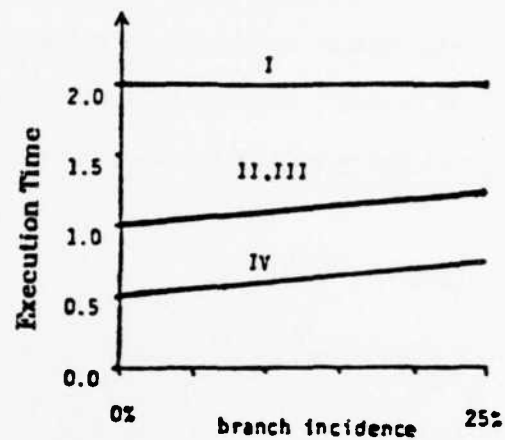
slot or data dependency optimization, respectively) is performed *earlier* than is needed in a sequentially executing program. Useful work may be done in the interim if the optimizing compiler can find an instruction to put into the empty slot. This is the goal of the *delayed jump scheme* [2]. This reordering is not easily done for conditional jumps for which branch prediction techniques must be utilized. Data from the VAX-11/780 indicate that conditional branches constitute 7% to 17% of the dynamic instruction count [5]. This penalty may be considered acceptable unless the number of pipelining levels is high. Stack machines are less flexible in terms of code reorganization; for this reason only register-based machines will be considered.

Conditional and unconditional branches, whether absolute or relative, constitute less than 25% of typical programs. Subsequent to optimization, unfilled branch slots for the MIPS processor vary from 3% to 24% (for a single slot per branch), and 21% to 50% (for two slots per branch) [6]. Results of this optimization vary depending on the programming environment and compiler technology. The IBM 801 compiler "...is able, generally, to convert about 60% of the branches in a program into the execute form." [7]. Figure 7(a) illustrates the effect of unoptimized branches on overall performance. In (b) this is compared among the four timing schemes. Estimated results of optimization are presented in (c), assuming the worst case MIPS unfilled branch slot incidence mentioned above (24% and 50%). As expected, the overhead of unfilled branch slots increases with pipelining. This overhead need not be directly proportional to the number of branches; the graph indicates an upper bound for convenience of discussion.

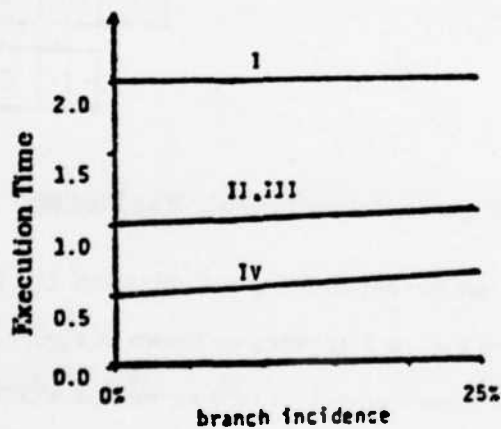
Dependencies arising from LOADs and branches are similar, with the exception that the former refers to data memory, and the latter refers to instruction memory. Such dependencies are inherently reduced with a register-based architecture. This is because the frequency of LOADs is reduced by depending



(a) Effect of slot overhead



(b) Comparison among pipeline schemes



(c) Comparison after optimization

Figure 7: Dependency Overhead and Optimization

mainly on local register storage of operands. It is expected that optimized data dependency overhead will be less than that for branches, since the dynamic LOAD count is typically less than 15%, which is observed for Quicksort [2]. Performance overhead of data dependencies may be determined with the aid of Figure 7.

Execution time overhead due to dependencies is also accompanied by a

corresponding increase in code size due to the NOOP instructions. Dependency optimization significantly reduces this overhead by replacing NOOPs with useful instructions. Elimination of remaining NOOPs may be accomplished by encoding wait states in the instructions responsible for the dependencies.

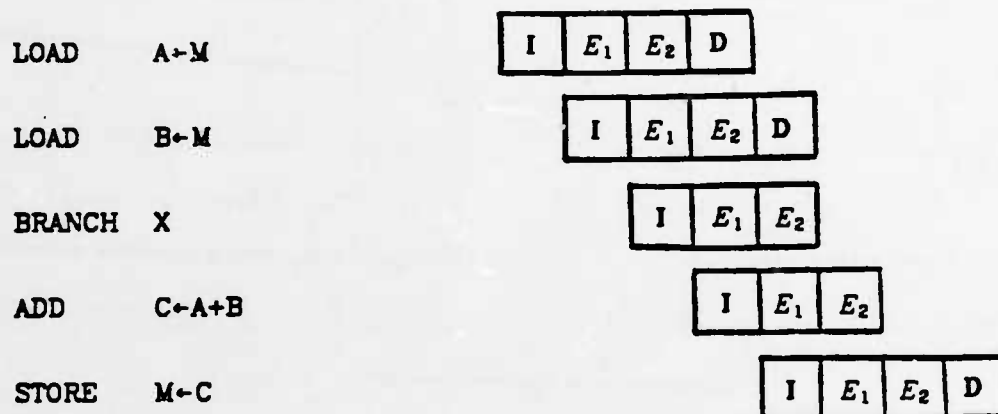


Figure 8: Reorganized Code for Four-Way Pipeline

After code reorganization for dependency optimization, the four-way pipelined instruction sequence of Figure 5 appears as shown in Figure 8. This example includes a performance improvement of 80% as well as a 37% reduction in code size.

This code optimization at the level of the machine cycle is important for highly pipelined machines. Complex instructions cannot be reordered at this level, and as a result perform poorly. Discussion of this issue for the MULTIPLY instruction is given in [8].

### Pipelining Datapath Modules

Pipelining may also be applied at the submodule level within the datapath. For example, the ALU and register file may each exploit several levels of concurrency. Dependencies have been investigated as one side-effect which limits

performance of a pipelined system. Additional costs of pipelining are the additional storage elements and associated clocks needed to hold intermediate results. A high degree of pipelining entails more circuitry for these functions. A consequence may be extra delay in the throughput of all instructions. This delay results from the propagation time through the added storage elements and data skewing, as well as the required clock setup time for latching the intermediate results. This overhead can be significant in a highly-pipelined module, such as the parallel adder example in [9]. Datapath pipelining will be considered in more detail in Chapter 4. Typically, pipelining in a datapath module is employed only to the degree that it helps to alleviate a severe bottleneck in system performance. The degree of attainable pipelining is then determined by the slowest link in the system which cannot be improved; often this is the I/O cycle. The following chapter will address I/O-limited performance issues.

## References

- [1] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 29-32, March 1982.
- [2] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer," Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.
- [3] S. Magar, E. Caudel, A. Leigh: "A Microcomputer with Digital Signal Processing Capability," International Solid-State Circuits Digest of Technical Papers, pp. 32-33, February 1982.
- [4] J. Hennessy, N. Jouppi, F. Baskett, J. Gill: "MIPS: A VLSI Processor Architecture," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, October 1981.
- [5] D.W. Clark and H.M. Levy: "Measurement and Analysis of Instruction Use in the VAX-11/780," Proceedings of the 9th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 9-17, March 1982.

- [6] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.
- [7] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.
- [8] M. E. Hopkins, "Compiling High Level Function on Low Level Machines," Proceedings of the International Conference on Computer Design, ICCD '83, pp. 617-619, Oct.-Nov. 1983.
- [9] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, vol. c-31, no. 3, pp. 260-264, March 1982.

## CHAPTER 3:

# LOCAL MEMORY TRADEOFFS

A fundamental limitation to processor performance is set by the ratio of the amount of memory traffic and the available I/O bandwidth. The bandwidth limit for a given technology is set by area and power constraints. Only a limited number of I/O pads with their associated driver circuits can be placed on the chip periphery. Wire bonding technology has not followed in the footsteps of the shrinking transistor; pad size has remained constant over the years. Power dissipation of I/O drivers is determined by a delay-power product, because the off-chip loading is primarily capacitive. Multiplexing the pads for several I/O transactions per cycle requires a faster settling time, and hence greater power dissipation.

Memory traffic consists of two classes of information: instructions and data. Several options are available for reducing either component. At a high level, the set of machine instructions may be designed to include powerful constructs which are equivalent to many simple instructions. This has been done traditionally for large mainframe computers. There is much debate, however, with regard to its use in VLSI implementation. Advocates of the simpler Reduced Instruction Set Computer (RISC) maintain that, within the constraints of single-chip implementation, a complex instruction set is a poor use of limited chip resources [1]. Microprocessors to date have devoted the majority of their die



area to instruction microcode ROM. RISC implementations utilize this area to provide more local memory. Use of local memory keeps much of the needed data local to the processor and allows data traffic to be reduced. A register-based machine, for example, can store frequently-used operands in a fast, multiple-port register file.

Register allocation is performed by the compiler and requires no hardware overhead. It is performed independently for each subroutine, thus procedure calls require separate register banks or blocks of local memory. Register contents may have to be swapped out of local memory in order to make room for the next procedure. The I/O overhead entailed is costly and may actually increase data traffic over that required by off-chip operand storage. In order to overcome this performance degradation, the RISC I microprocessor organizes its local memory as multiple register banks, with each bank supporting a different procedure level [1]. In addition, adjacent banks overlap partially in order to facilitate parameter passing among subroutines. This approach drastically reduces data I/O traffic.

A multiprogramming or multitasking environment puts even more stringent demands on the performance of local memory. During each context switch a new register bank must be made available for the next executing program. In the case of a single register bank, its contents must be saved during every context switch. Multiple banks reduce this overhead by allowing context information from several programs to reside on chip. Since multitasking may be interrupt driven, arbitrary switching to any other process must be allowed. This contrasts with procedure level changes, where a stack organization will suffice. Microprocessors implementing context switching support include the Fujitsu FSSP (4 banks) [2] and the Siemens SAB 80199 (8 banks) [3].

All implementations with multiple register banks for procedure or context

switching support must accommodate register overflows. When the number of banks is exceeded, some swapping to external memory is required in order to make room on chip. When the capacity of a given bank is exceeded, external memory storage must be used, to store additional operands.

The alternative to the approach discussed above is a pure memory-to-memory architecture. With this scheme all data are stored in external memory, and no saving or restoring of registers is necessary upon a procedure call or context switch. On the other hand, all operand manipulations require data load and store operations to be performed. The above mentioned data I/O bottleneck, and the resulting greater latency compared to that of the register file architecture, may reduce performance. An example of a memory-to-memory microprocessor is the TMS 9995 [4].

The relative merit of the memory-to-memory approach versus a register-based machine depends on the programming environment and memory performance. Additional data traffic of the memory architecture must be compared to that incurred in a register machine when the procedure nesting depth or number of processes exceed the available number of register banks, as well as that occurring when capacity of each bank is exceeded.

Clearly, an infinitely large local memory is desirable since it can reduce off-chip data traffic to zero. Thus, the architects would like to have as much on-chip memory as possible. However, if the local memory is too large, it will also be slower, and system performance will be degraded. This chapter analyzes these tradeoffs.

### Local Memory in RISC II

The RISC II is the second in a series of 32-bit, NMOS microprocessors developed at U.C. Berkeley [5]. The RISC instruction set consists solely of single

register-to-register operations [6]. This simple and regular implementation reduces control complexity, chip area, and design time, and it simplifies implementation of pipelined execution [7]. The simple RISC instruction set is an easier target for highly optimizing compilers than is a complex instruction set [8]. Proper optimization can also reduce dependency overhead inherent in pipelined implementations [9], thus making more effective use of the available datapath bandwidth.

A drawback of such an instruction set is that it requires higher memory bandwidth for fetching these instructions. Because the instructions are simpler, it often requires several of them to synthesize a complex instruction. This increases overall code size. On the other hand, the RISC microarchitecture includes support for subroutine call and return, one of the most time-consuming operations in typical high-level language programs for machines which keep variables in registers [6]. The number of register saves and restores is reduced by employing a local memory organized as multiple register banks. A new bank is allocated whenever a procedure is called. The banks represent stack levels, so that register save or restore need be performed only during stack overflows or underflows.

RISC II includes eight register banks, or windows, one of which is reserved for interrupt processing. At any one time there are ten registers local to the present procedure level. Additionally, there are six "high" and six "low" registers which are shared by adjacent procedure levels; these are used primarily for passing parameters and results between procedures. Each window swap (for save or restore) involves sixteen registers: the ten locals and one set of overlaps. Ten global registers are accessible from any procedure level, thus a total of 32 registers are addressable from any procedure.

Table I and Figure 1 show the relative execution time and performance of two C programs, "Tower of Hanoi" and "Puzzle", versus the number of windows on the chip. These results are based on earlier studies of procedure behavior and register file management overhead for RISC [10,11]. Both benchmarks nest to a depth of twenty. However, "Tower" has a very high rate of procedure calls and returns (19%) and thus makes intensive use of the multiple windows. Performance improves steadily through the use of, say, seven windows. "Puzzle", on the other hand, performs well with only one or two windows; it has only 0.7% calls and returns.

WINDOWS	1	2	3	5	7	9	$\infty$
TOWER 19% Dynamic Call & Return	7.08	3.02	2.52	1.38	1.10	1.02	1.00
PUZZLE 0.7% Dynamic Call & Return	1.17	1.02	1.00	1.00	1.00	1.00	1.00

TABLE I: Normalized RISC II Execution Time  
(relative to case of infinite windows)

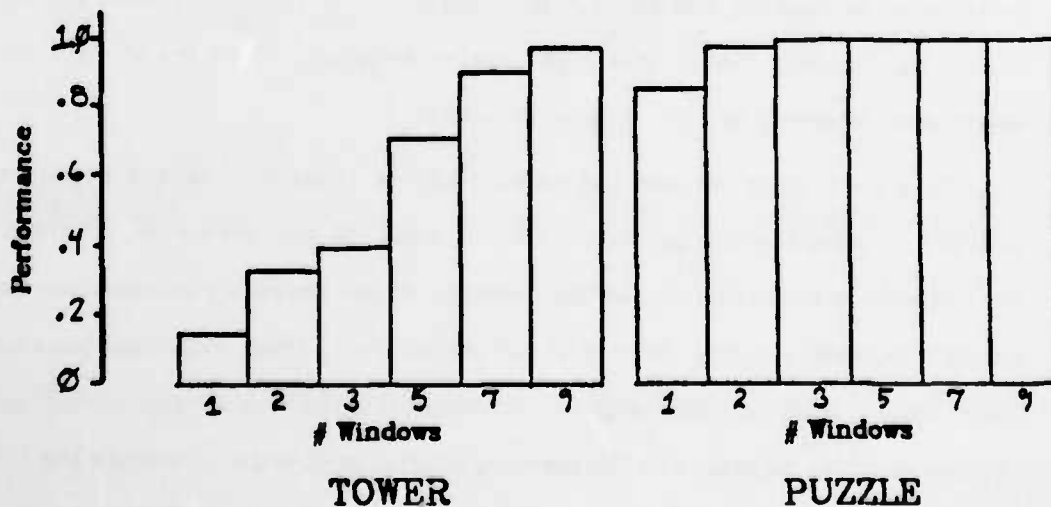


Figure 1: Normalized Performance of RISC II

Typical programs have a procedure call or return every twenty instructions, so the benchmarks shown here represent extremes [12,13]. In consideration of limited chip resources, a careful analysis of the program environment is desirable. If few procedures are used, a smaller local memory allows resources to be utilized for performance improvement in other areas.

### Cost of Fixed-Size Window Swaps

The cost of register window overflow is determined by two factors: the overhead of servicing the interrupt caused by the overflow, and the cost of the actual data transfer between the register file and external memory. The RISC II microprocessor incurs a penalty of about thirty instructions for the window overflow/underflow interrupt routine. The single I/O bus implementation supports one memory access per cycle, which means that each load or store for the window swap takes two cycles. With sixteen registers per window, a total of 32 cycles are required. Although each window swap is costly, overflow/underflow occurs infrequently if there is a sufficient number of windows on chip. However, reducing local memory size below a program-dependent limit degrades performance significantly due to this high cost of swapping. With fewer windows, better swap interrupt and I/O support is crucial.

Since each swap utilizes the same protocol (sixteen adjacent registers swapped to/from the current window) better data I/O support can be provided. For example, a single instruction may provide all the necessary information for multiple register copying. Then it is not necessary to fetch individual Load or Store instructions for each register transfer. Furthermore, only a starting address needs to be passed to the memory controller in order to initiate the 16-word move. Two data words may then be passed on the bus each machine cycle. Compared to the present scheme, with one data word every two cycles interleaved with instruction fetching, throughput is increased by a factor of four.

At compile time, the dynamic procedure nesting profile is not known. Therefore the compiler cannot anticipate window overflows in a multiple-window implementation. For this reason, overflows must be detected on chip. It is the cost of handling this interrupt which accounts for thirty instructions in RISC II. For a processor with a single window, the compiler can anticipate the swaps and this overhead can be reduced. Every executed call or return requires a save or restore operation, respectively.

Table II presents RISC II execution time as a function of data I/O bandwidth and local memory size. The cost of each swap includes the thirty cycles for interrupt overhead, as well as the sixteen data word transfers. Since swap overhead for "Puzzle" is small, only "Tower" is considered here.

WINDOWS	1	2	3	5	7	9	$\infty$
One-Half Data I/O Per Cycle	7.08	4.91	3.95	1.74	1.19	1.05	1.00
Single Data I/O Per Cycle	4.04	3.90	3.19	1.55	1.14	1.03	1.00
Dual Data I/O Per Cycle	2.52	3.39	2.81	1.46	1.12	1.03	1.00
Swapping Interrupt Overhead	0.00	1.89	1.43	0.36	0.09	0.02	0.00

TABLE II: Execution Time for "Tower" with Varying Swap Bandwidth  
(includes interrupt overhead for multiple window cases)

Performance penalty due to interrupt overhead is illustrated by the shaded area in Figure 2. As before, seven or more windows are desirable for high performance, regardless of the level of data I/O support. This is because the interrupt overhead is so high. An exception is the single window case with dual data I/O per cycle, which is seen to provide better performance than register files

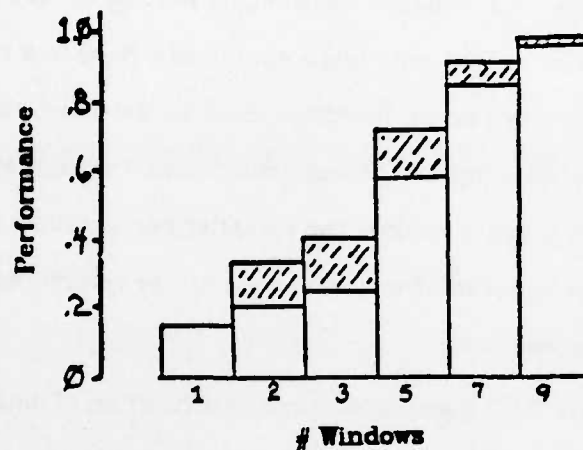


Figure 2: Performance with Overhead of RISC II Swap Interrupt  
("Tower" Benchmark, half data I/O per cycle)

with two or three windows. With a large local memory, efficient interrupt support is not so important since few swaps occur. With few windows, though, it can be crucial. The remainder of this chapter will not consider the interrupt overhead; it is assumed that the area freed by reducing local memory size may be dedicated toward better interrupt support, and that overall swapping cost is dominated by the register traffic.

### Improved Swapping Strategies

Thus far, we have only considered fixed-size window swaps for which all registers are transferred to/from memory. This scheme is attractive due to its simplicity and ease in providing higher swap bandwidth. However, such a scheme swaps all registers in the window, whether they were used or not. A study of several C programs has determined that on the average only four registers are used per procedure in RISC [10]. Therefore, the fixed-swap scheme performs four times the number of necessary save and restore data transfers.



### Register Usage Record

In order to keep track of the registers actually used, a "dirty bit" may be employed. During each register write, a bit is set to indicate a register that needs to be saved when the window gets swapped. Swaps thus vary in length, depending on the number of bits set. The increased hardware complexity necessary to support such an approach, however, is undesirable.

An alternative is to utilize a single-word register usage mask for each window. Each bit which is set in the mask indicates usage of a specific register. A stack of such masks must be maintained on-chip for resident windows. During a window overflow, the appropriate mask is stored with the window contents. Additional logic is required in order to encode and decode the mask and provide for a mask stack.

A single window implementation does not require this hardware. The compiler can insert code before each call in order to save the registers used in the current window. Restoring registers after a return can be done on demand by the compiler. Since not all registers may need to be restored, further reduction in I/O is anticipated.

Save overhead may also be eliminated by performing a data memory write in parallel with all register file writes. This "store-through" scheme requires a dual-bus microarchitecture which can fetch an instruction and perform a data access in each cycle, such as the TMS 320 [14] or MIPS [15] microprocessors. Overall, swap overhead may then be reduced by more than a factor of eight.

### Variable Window Size

Although we have described alternative strategies for local memory management, we have not yet addressed effective use of the on-chip memory

area. The above schemes reduce data traffic and off-chip register save space by a factor of four, but on-chip memory still attains only 25% utilization. If the register file windows can vary in size, such a waste of resources can be avoided.

For a variable-size window scheme, "bank" and "window" no longer need to be synonymous. The register file may be divided into fixed-size banks for regular and efficient swapping. Several procedures may reside within a single bank of, say, 16 or 32 registers; they may also span bank boundaries. This scheme requires additional hardware, in the form of pointers for each procedure domain, and an adder to calculate the physical addresses of the registers. Further details of variable-size window schemes may be found in [1].

WINDOWS	1	2	3	5	7	9	$\infty$
Full-Bank Register Swaps	4.04	2.01	1.76	1.19	1.05	1.01	1.00
Partial Register Swaps	1.76	1.25	1.19	1.05	1.01	1.00	1.00
Partial Swaps with Store-Through	1.38	1.13	1.10	1.03	1.00	1.00	1.00
Variable Size with Full-Bank Swaps	2.01	1.21	1.08	-	-	-	1.00

TABLE III: Execution Time for "Tower" with various Swap Schemes  
(one data I/O per cycle assumed for all cases)

Performance comparison of these schemes is presented in Table III. All cases assume single data I/O per cycle and four registers per procedure. Interrupt overhead, which occurs for the multiple window and variable size schemes, is not included here. The variable window scheme is assumed to utilize two equal-size banks, with total register count being the number of windows indicated in the table times sixteen. One of these banks is swapped during an overflow or underflow. Total number of registers is sixteen times the number of

windows indicated in Table III. Significant performance improvement is observed for implementations with few windows using these alternative swap schemes. By using more efficient swapping strategies, high performance may be attained with less chip area dedicated to local memory.

### Register File Delay

Up to this point, only I/O limited performance has been discussed. From the designer's point of view, attention should also be focused on datapath bandwidth. Especially for RISCs, where each execution cycle consists of a uniform register-to-register operation, the datapath cycle time determines maximum system performance. The machine cycle of the RISC II consists of a dual-port register read, followed by an ALU or shift operation; these latter operations overlap the register write of the previous instruction and bitline precharge of the next instruction. The machine cycle is then limited directly by the register file read-write-precharge cycle time.

The register file cycle time increases with local memory size and depends on several design parameters. Read delay consists of two components: wordline assertion, and bitline discharge (Figure 3). The wordline, or addressing, delay is proportional to the gate capacitance loading of the access transistor. This delay then is linearly proportional to word size. In technologies where the wordline itself is the dominant resistance in addressing delay, such as with a polysilicon wordline, this delay follows the square of word length. Periodic buffering of the address (as for the dynamic ripple carry ALU) is necessary to reduce this delay to a linear function of word size. Bitline discharge delay increases with the product of the resistance of the wordline access transistor and the bitline loading capacitance. This delay then increases proportionally with memory word capacity.

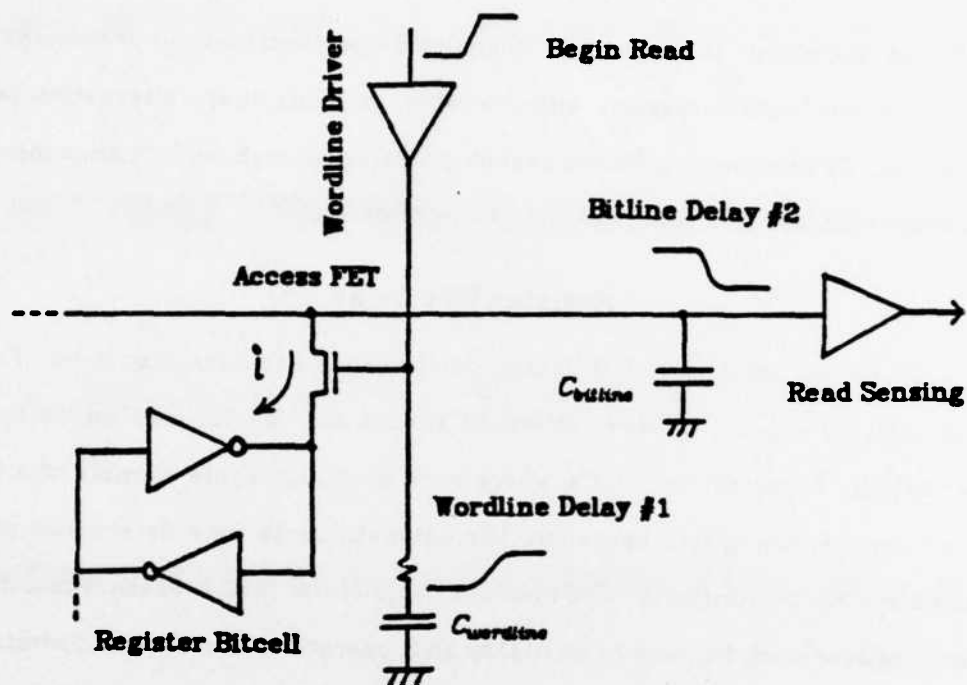


Figure 3: Register File Read Delay

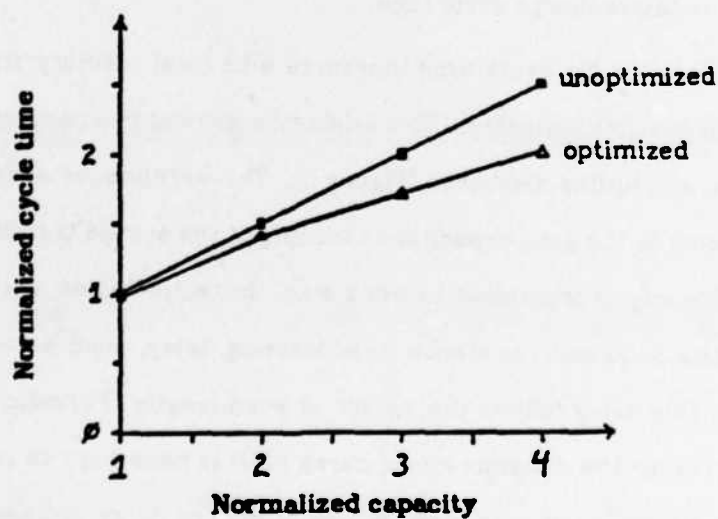


Figure 4: Cycle Delay versus Local Memory Capacity

Total register delay is therefore expected to increase proportionally with local memory size. However, some optimization is allowed by the wordline

access transistor. Reducing its width allows faster addressing at the cost of increased bitline discharge delay. Optimal performance is attained when both delays are equal [18]; this implies an access transistor size (and hence register cycle delay) which is proportional to the square root of the memory capacity. The write and precharge delays are then affected similarly. As a result, datapath cycle time increases with the square root of word length or word capacity of the register file (Figure 4). This effect must be taken into account to obtain a more realistic performance estimate for various local memory sizes.

WINDOWS		1	2	3	5	7	9
Normalized RISC II Register Cycle Time (ignoring swaps)		1.00	1.22	1.41	1.73	2.00	2.24
Full-Bank	$\frac{1}{2}$	7.08	3.68	3.55	2.39	2.20	2.28
Register Swaps with Varying Data I/O per cycle	1	4.04	2.45	2.48	2.06	2.10	2.26
	2	2.52	1.84	1.95	1.90	2.05	2.24
Partial Register Swaps		1.78	1.53	1.68	1.82	2.02	2.25
Partial Swap with Store-Through		1.38	1.38	1.55	1.78	2.01	2.24
Variable Size with Half Bank Swaps		2.01	1.48	1.52	-	-	-

TABLE IV: Datapath Bandwidth Limited Execution Time  
(smaller local memory is faster; using "Tower" benchmark)

Table IV includes the effect of variable register cycle delay. The partial register swap schemes using a single window yield the best performance, only rivaled by the variable size scheme with its additional hardware complexity. A single window, fixed-swap implementation with dual data I/O per cycle approaches the performance of the seven window RISC II with half data I/O per cycle. Execution time for "Puzzle", with little swap overhead, follows the

register cycle time dependence with memory size; it executes nearly twice as fast with one window as it does with seven. Inclusion of register file delay then has a major impact on the relative merits of the swapping schemes. Of course, the smaller local memory implementations have higher datapath bandwidth, so a similarly faster external memory is required in order to realize this performance. Local memory size is traded off against memory bandwidth requirements.

Chip design tradeoffs in VLSI must be made using both architectural and circuit design considerations. One measure of the cost of local memory, increased delay, yields two minimum execution time solutions: I/O limited, and datapath bandwidth limited. Because of the limited number of pads that can be placed on a chip, memory I/O is a severe bottleneck in system performance. For this reason, a large local memory was chosen for RISC II. Presently, memory speed is increasing, making datapath bandwidth a more critical limit to system performance. In the future, increased chip resources will make possible a greater local memory hierarchy [17]; I/O bandwidth may then be replaced by datapath bandwidth as the primary factor limiting system performance.

### References

- [1] M.G.H. Katevenis: "Reduced Instruction Set Computer Architectures for VLSI," Doctoral Dissertation, Computer Science Division, University of California, Berkeley, 1983.
- [2] N. Inui, H. Kikuchi, T. Sakai: "16-bit C-MOS Processor Packs in Hardware for Business Computers," *Electronics*, pp. 182-186, June 16, 1981.
- [3] J. Gosch: "Microprocessor does Multitasking in Real Time," *Electronics*, pp. 71-71, Nov. 3, 1982.
- [4] J. Schabowski: "Tough Control Tasks Take 16 Bits," *Electronics*, pp. 91-94, Dec. 18, 1980.
- [5] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson, C.H. Séquin: "The RISC II Micro-Architecture," *Proceedings of the IFIP TC10/WG10.5 International*

Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.

- [8] D.A. Patterson, C.H. Séquin: "A VLSI RISC," IEEE Computer, vol. 15, no. 9, pp. 8-21, September 1982.
- [7] J. D. Wright: "Relation of Microcode to Future Machine Design," COMPCON Digest of Papers, pp. 104-106, March 1983.
- [8] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.
- [9] T. Gross: "Code Optimization Techniques for Pipelined Architectures," COMPCON Digest of Papers, pp. 278-285, March 1983.
- [10] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames," CS 292R Final Class Report, Computer Science Division, University of California, Berkeley, Spring 1980.
- [11] Y. Tamir, C.H. Séquin: "Strategies for Managing the Register File in RISC," IEEE Transactions on Computers, vol. c-32, no. 11, November 1983.
- [12] D.R. Ditzel, H.R. McLellan: "Register Allocation for Free: The C Machine Stack Cache," Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, pp. 48-56, March 1982. (ACM: SIGARCH CAN vol. 10, no. 2, SIGPLAN Notices vol. 17, no. 4)
- [13] D.W. Clark, H.M. Levy: "Measurement and Analysis of Instruction Use in the VAX-11/780," Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, pp. 9-17, March 1982.
- [14] S. Magar, E. Caudel, A. Leigh: "A Microcomputer with Digital Signal Processing Capability," Proceedings of the International Solid-State Circuits Conference, pp. 32-33, February 1982.
- [15] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Proceedings of the Third Caltech Conference on VLSI, Computer Science Press, pp. 33-54, March 1983.
- [16] A.M. Mohsen, C.A. Mead: "Delay-Time Optimization for Driving and Sensing of Signals on High-Capacitance Paths of VLSI Systems," IEEE Journal of Solid-State Circuits, vol. sc-14, no. 2, pp. 231-239, April 1979.
- [17] D.A. Patterson, C.H. Séquin: "Design Considerations for Single-Chip Computers of the Future," IEEE Journal of Solid-State Circuits, vol. sc-15, no. 1, pp. 44-52, February 1980.



## CHAPTER 4:

# DATAPATH TIMING

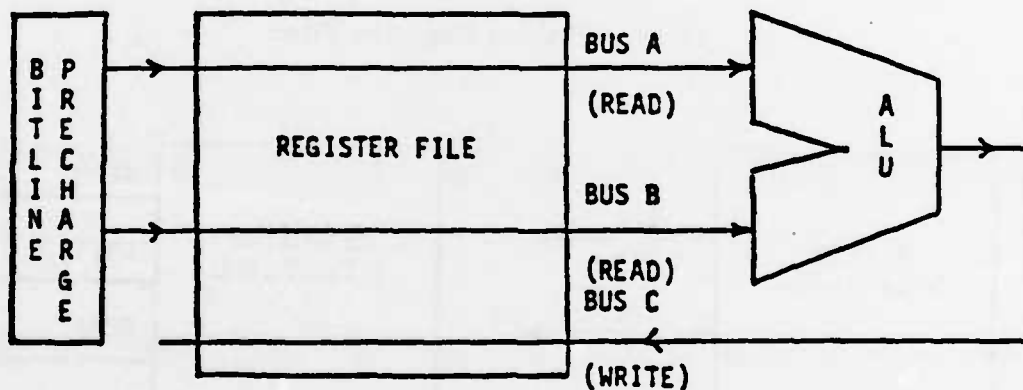
A critical issue for register-to-register machines is the register file organization and timing. A variety of bitline and wordline configurations yields a wide range of datapath bandwidth. The microarchitect must determine what silicon resources are required for each approach and study the tradeoffs between the various timing schemes. The purpose here is to review a variety of possible datapath timing schemes in order to develop an intuitive understanding of the tradeoffs involved.

Various datapath designs are compared here in terms of the speed at which register-to-register (R-R) operations can be executed. This determines the performance limit of simple, register-oriented machines, such as the RISC I [1], MIPS [2], and the 801 [3]. Regular instruction cycle timing yields simple, regular implementation of the control circuitry. Regularity of instruction execution permits pipelining without requiring complicated hardware pipeline interlocks.

The benefits of instruction pipelining may also be exploited by a micro-coded machine. Pipelining favors the use of regular, register-to-register microoperations, rather than the use of irregular, but fast, microcoded implementation of the relatively few dynamically executed, compiled complex instructions [4].

When viewed from the datapath, the actual instruction coding is not visible. Therefore no assumptions regarding the machine's instruction set are made in this chapter. Instruction coding is a higher level issue, for which tradeoffs can be assessed once a programming environment has been chosen.

For simplicity, overhead of I/O will be ignored. Performance limits due to off-chip communication were considered in the previous chapter. Program counter logic will not be considered explicitly; it may be encompassed for our purposes within the domain of the addressable register file. Therefore, ideal system speed, to be discussed in this chapter, is determined directly by the datapath bandwidth.



REGISTER READ	ALU OPERATION	REGISTER WRITE	BITLINE RESTORE
---------------	---------------	----------------	-----------------

Figure 1: Register-to-Register Timing Example

The fundamental datapath operations to be performed during each register-to-register cycle are shown in Figure 1. They include reading two operands from the register file, performing an ALU or shifter operation, and writing the result back into the register file. In NMOS implementations, the read

bitlines must be restored to a logic "1" prior to reading. This is necessary if the read bitline is dynamically precharged, and/or the bitline is used for both reading and writing through the same register cell port. This is to ensure the read value is valid, and that the read operation does not accidentally *write* into the cell.

A critical, limiting factor in determining allowable concurrency is the register file organization: three of the four basic operations concern it. Various bitline (bus) and wordline (addressing) organizations will be considered in discussing timing schemes which exploit greater levels of concurrency than the sequential example above.

### Shared Bitline Register Files

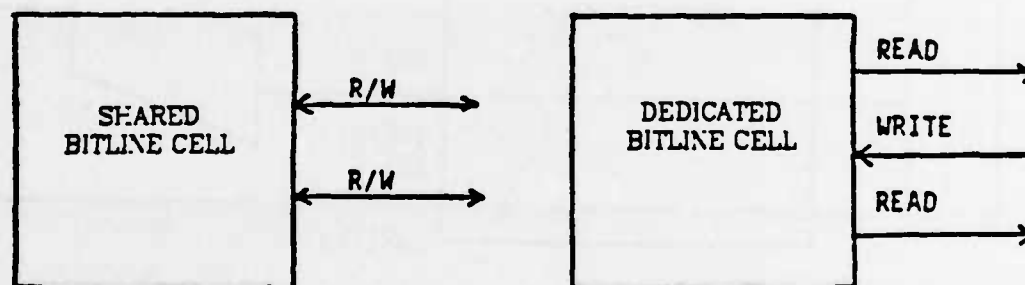


Figure 2: Shared and Dedicated Bitline 2-Port Cells

Of fundamental importance in determining allowable concurrency within a register file read-write cycle is the bitline arrangement. A *shared bitline* organization utilizes the same bitlines for both reading and writing. A *dedicated bitline* approach utilizes separate bitlines for reading and writing, allowing some overlap of these operations (Figure 2). Advantages of the shared bitline design include its economy of area, due to fewer bitlines and fewer transistors. This cell may also be faster, since its smaller size reduces loading on the wordlines. This helps make up for the reduced level of concurrency attainable with fewer

bitlines. Overall system timing for the shared bitline approach is identical to that in Figure 1.

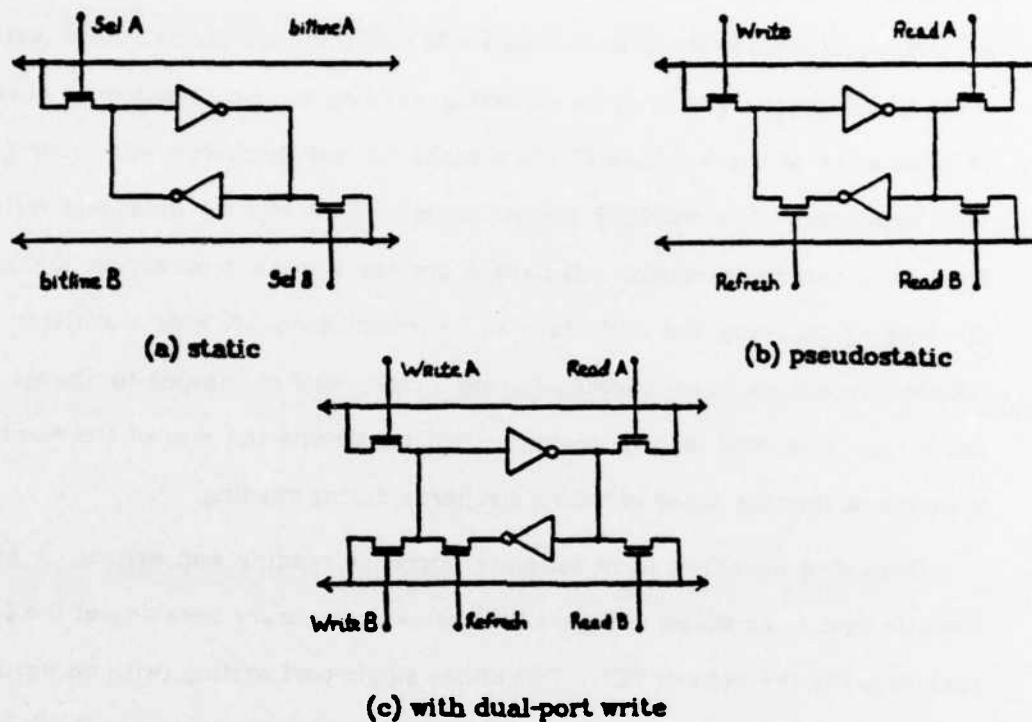


Figure 3: Shared Bitline Register Cells

Another design choice exists for the shared bitline cell, leading to two different structures. One may use shared (Figure 3(a)) or dedicated (Figure 3(b)) wordlines for the read and write operations. In the first case there is one set of wordlines, to be used for both reading and writing to common ports; writing is usually performed via complementary signals driving two ports. This is a derivation of typical commercial Random Access Memory (RAM) designs. The shared wordline approach leads to a fast and compact cell. Such a static design has been implemented in the RISC II, which requires a large, dual-port register file [5]; its symmetry was crucial in attaining a compact layout.

Care must be taken to ensure that reading onto a precharged bus will not

change the state of the shared wordline cell. As shown in Figure 3(a), each bit cell inverter may be considered to be transformable into a 2-input NOR gate by the addition of the wordline transistor. If the bus is initially discharged to ground, wordline access will behave as a NOR input and set the cell state (write). If the bus is precharged prior to accessing, reading will cause either no change (if both sides of the wordline FET are high), or bus discharge will occur (cell node grounded). The wordline transistor in this latter case acts as a voltage divider. A narrow transistor will have a greater voltage drop across it during discharging, allowing the cell state to be maintained. A wide transistor will reduce the voltage drop, allowing the high logic level of the bus to change the cell state. This *read disturb* problem then constrains the size of the wordline transistors, limiting speed of bitline discharge during reading.

Dedicated wordlines form separate ports for reading and writing. A pseudostatic design, as shown in Figure 3(b), allows temporary breaking of the feedback loop (by the refresh FET). This allows single-port writing (with no wordline FET voltage drop) and eliminates read disturb by breaking the feedback loop. The wordline transistor size is not constrained as before. In the case of a dual-port pseudostatic cell with two pairs of wordlines, two locations in the register file may be written simultaneously (Figure 3(c)). This scheme was chosen for the Caltech OM-2 [6], as well as the HP FOCUS chip [7]. In order to maintain data integrity, the refresh transistor must be clocked periodically. This is usually done every cycle, during the bitline restore phase.

A disadvantage of this design is its asymmetry, due to the refresh transistor and the single inverter drive for reading. The first inverter is not used for reading because its gate can only be charged to  $V_{DD}-V_T$  by the enhancement pass transistors. Overall cycle time then must include the worst case delay of discharging both bitlines.

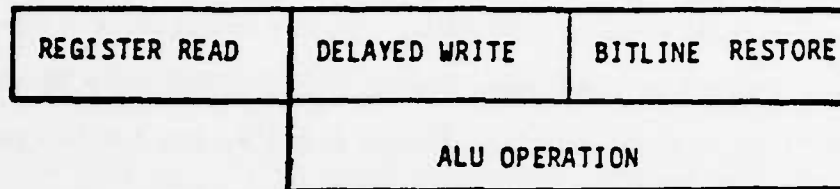


Figure 4: Timing of Delayed Write Scheme

In order to reduce the datapath cycle from four phases to three, the RISC II increased the level of pipelining and incorporated a *delayed write* scheme [8]. In effect, writing is delayed to overlap the ALU computation of the following instruction. This added level of pipelining is helpful as it allows greater time for interrupts to be detected without destroying the contents of the register file. Also, if the ALU delay is significant, it may overlap *both* the register write and bitline restore phases (Figure 4).

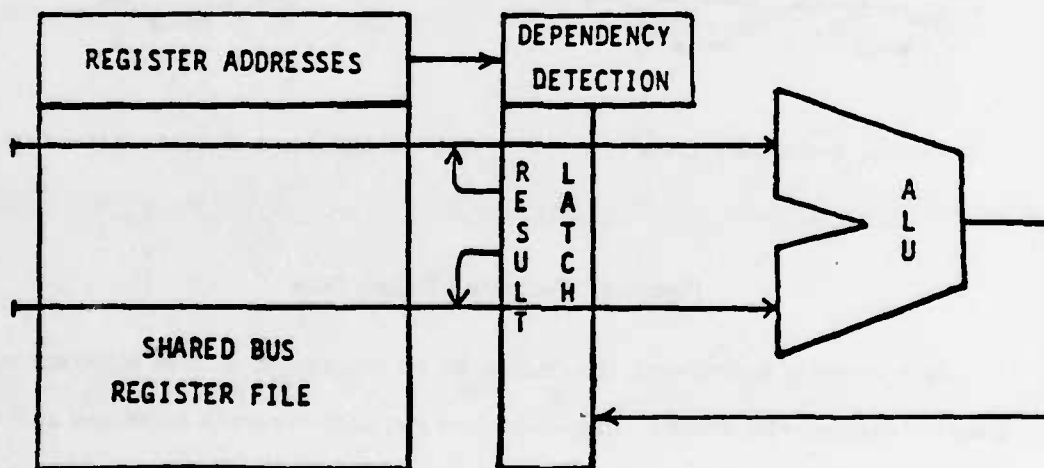
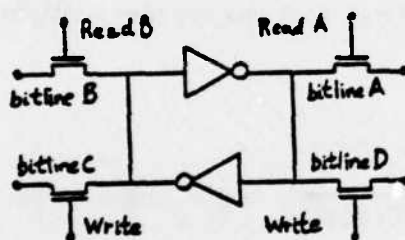


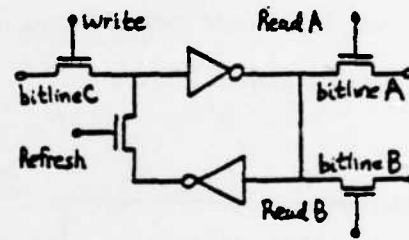
Figure 5: Delayed Write with Internal Forwarding

Some performance degradation might result from this scheme due to *data dependencies*. The result of a computation is not available in the register file for the read phase of the following instruction. This is a consequence of this pipelined implementation with the delayed write. The problem was solved in the case of RISC II by detecting data dependencies and forwarding the data through a temporary register to the ALU or shifter. This *internal forwarding*, or *chaining* technique allows the data in this register to *override* the result of the register file read (Figure 5). This technique is transparent to the programmer or compiler writer. Such an approach is routinely used to increase performance of highly-pipelined computers, such as the CRAY I.

### Dedicated Bitline Register Files



(a) Static Dedicated Bitline Cell



(b) Pseudostatic Dedicated Bitline Cell

Figure 6: Dedicated Bitline Cells

As previously mentioned, the dedicated bitline design utilizes separate bitlines for reading and writing. Implicitly, this requires separate wordlines as well to guarantee independence of read and write operations (see Figure 6). This structure supports a higher level of concurrency and therefore may be desirable for a high-speed datapath. Restoring of the bitline may overlap the writing of the



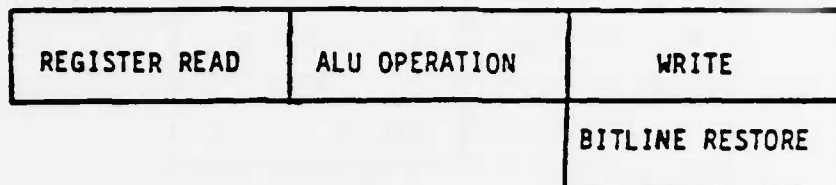


Figure 7: Timing of Dedicated Bitline Datapath

cell since it uses separate control and data lines. This makes possible the three-phase timing of Figure 7. Such an approach has been used in the RISC I [8] and the Matsushita MN1613 [9]. This scheme, however, will be slower than the previous approach (shared bitline with delayed writing) if the ALU delay is greater than that of the bitline restore. This, in conjunction with the cell area difference, makes the three-phase dedicated bitline scheme discussed here undesirable for large register files.

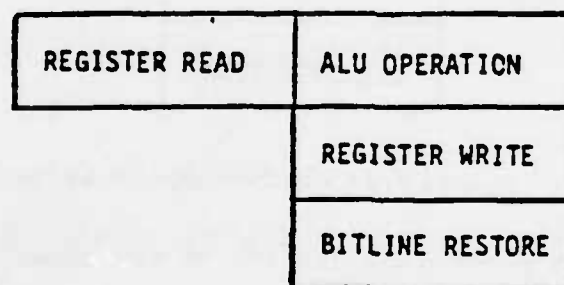


Figure 8: Timing of Dedicated Bitline with Delayed Write

In order to increase the concurrency, the delayed-write scheme may be used. Timing of the ALU operation, register write, and the bitline restore all overlap (Figure 8). Internal forwarding logic is necessary to eliminate data dependency problems as before. Forwarding is performed in parallel with the register write.

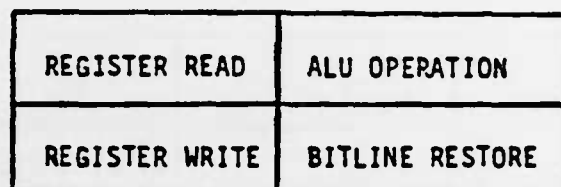


Figure 9: Overlapped Read/Write Scheme Timing

Alternatively, the read and write operations may be overlapped, as shown in Figure 9. Dependency detection logic is again required, and internal forwarding is performed in parallel with the register write.

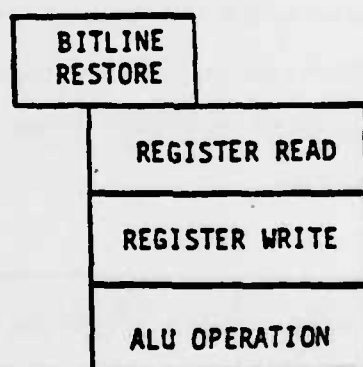


Figure 10: Delayed Write with Overlapped Read Timing

For even higher performance, two sets of data dependency logic are required. The first forwards the result of an ALU or shift operation to the data read register of the following instruction. The second forwards this data, as it is being written into the register file, to the read register for the instruction after that. This allows the greatest concurrency, as shown in Figure 10.

In order to combine the register read and bitline restore in a single phase, the restore must be initiated early enough during the read phase so that it overlaps the addressing delay. At the time the read wordlines are driven to a logic

"1", the bitlines must be precharged above the bit cell logic threshold in order to eliminate writing into the cell. Precharge may continue, overlapping wordline delay so that adequate noise margins are maintained. Alternatively, current sensing may be used, in which case the bitline voltage remains constant. This technique has been utilized in MOS ROMs [10].

System throughput for this single-phase timing scheme may be quadruple that of the original four-phase sequential example at the beginning of this chapter. This performance increase is achieved by maximizing module usage in each phase, in tune with effective chip resource utilization.

The treatment of datapath timing and register file organization has been very simplistic in this chapter. Since the bit cells must be designed uniquely for each timing scheme, their area will vary. This has a varying impact on chip resources and on cycle delay time. A more detailed analysis is thus required for the selection of an optimal register cell and timing scheme. This will be discussed in more detail in Chapter 8.

## References

- [1] D.A. Patterson, C.H. Séquin: "RISC I: A Reduced Instruction VLSI Computer." Proceedings of the 8th Symposium on Computer Architecture, ACM SIGARCH CAN, pp. 443-457, May 1981.
- [2] J. Hennessy, N. Jouppi, F. Baskett, J. Gill: "MIPS: A VLSI Processor Architecture," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, October 1981.
- [3] G. Radin: "The 801 Minicomputer," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, ACM SIGARCH CAN, pp. 39-47, March 1982.
- [4] J. D. Wright: "Relation of Microcode to Future Machine Design," COMPCON Digest of Papers, pp. 104-106, March 1983.
- [5] R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, C.H. Séquin: "Datapath Design for RISC," Proceedings of the Conference on Advanced Research in

VLSI, Massachusetts Institute of Technology, pp. 53-62, January 1982.

- [6] C.A. Mead, L.A. Conway: *Introduction to VLSI Systems*, Addison Wesley Publishing Co., 1980.
- [7] J. Beyers, L. Dohse, J. Fucetola, R. Kochis, C. Lob, G. Taylor, E. Zeller: "A 32-Bit VLSI CPU Chip," *IEEE Journal of Solid-State Circuits*, vol. sc-16, no. 5, pp. 537-541, October 1981.
- [8] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," *VLSI Systems and Computations*, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," *VLSI Design*, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and *Computer Architecture News (ACM SIGARCH)*, vol. 10, no. 1, pp. 28-32, March 1982.
- [9] H. Kadota, S. Ozawa, K. Kawakami: "A New Register File Structure for the High-Speed Microprocessor," *IEEE Journal of Solid State Circuits*, vol. sc-17, no.5, pp. 892-897, October 1982.
- [10] J. Wong, P. Siu, M. Ebel: "A 45ns Fully-Static 16K MOS ROM," *Proceedings of the International Solid-State Circuits Conference*, pp. 150-151, February 1981.

## CHAPTER 5:

# ALU DESIGN TRADEOFFS

Traditionally, evaluation of different adder schemes has been carried out with the assumption of a fixed gate delay. Such a straightforward comparison is permitted by low levels of integration, using SSI parts. These parts are designed to accommodate a wide range of capacitance loading due to off-chip wiring. As a result, delay exhibits little dependence on the loading capacitance typically encountered [1]. The designer calculates circuit delay by simply determining the number of gates in the critical path.

The custom nature of VLSI, on the other hand, gives the designer more freedom to optimize performance. Dynamic logic and bootstrapping techniques can be used to increase performance. Under this variety of approaches, gate delays can no longer be considered constant. Comparison of adder performance based on the fixed delay model is inadequate for VLSI implementation.

This chapter will begin with a review of adder design strategies. Initially a gate-level view will be used in order to simplify understanding. It is also directly applicable to fixed gate delay analysis. This will be followed by a discussion of design in NMOS using dynamic logic and bootstrapping. Finally, different carry schemes will be evaluated for both the fixed delay and NMOS implementations.

### Adder Design at the Gate Level

An example of a single-bit cell of a full adder is shown in Figure 1. Three delays exist: input translation, carry calculation, and sum generation. The

translation and sum delays are constant; they each consist of a single gate delay. The carry delay, however, is cumulative since its calculation is dependent on the result from the previous bit cell. The carry output of the most significant bit is thus dependent on all the previous stages. Overall carry delay will vary with the method used for its calculation, as well as with the number of bits  $N$ . For this reason we will focus on the circuitry that calculates the carry.

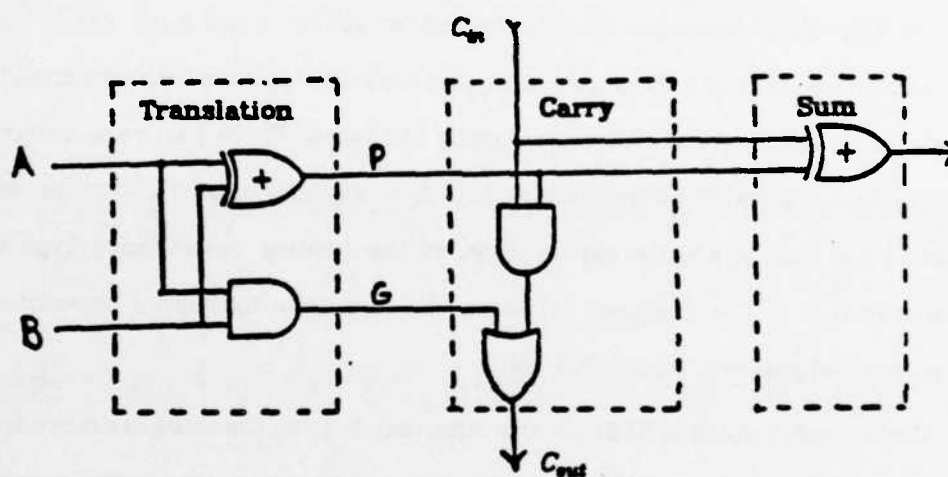


Figure 1: Full Adder Cell

### Ripple Carry

The simplest adder scheme utilizes a ripple carry as shown in Figures 1 and 2. For an  $N$ -bit adder, the carry propagates, or ripples, across  $N$  stages. Each stage consists of 2 gate delays, so the total carry delay for an  $N$ -bit adder is  $2N$ . Advantages of this design include minimal gate count, as well as regularity and short wire length for implementation in VLSI.

The ripple carry approach is used for small word sizes or in applications where speed is not critical. An 8-bit ripple adder was chosen for the Intel 8080 8-bit microprocessor because the regular, compact layout had less parasitics and was actually faster than a lookahead approach [2]. The Motorola 68000 used

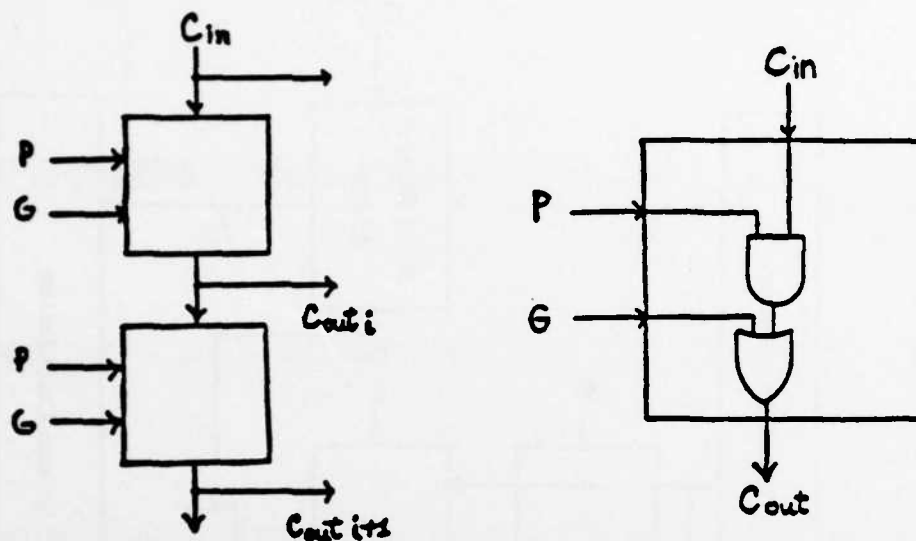


Figure 2: Ripple Adder Scheme (2 bits shown)

a 16-bit ripple adder because it was found to be faster than a lookahead adder with the same amount of power dissipation [3]. Ripple adders were also used for the 16-bit Caltech OM-2 [4] and the 32-bit RISC II [5] implementations, mainly because of their small chip area and short layout time.

Methods of reducing ripple carry delay are presented in [2]. Using an increased fan-in of 4, the delay can be reduced to an average of 4 gate levels for each 3 bit group by propagating multiple intermediate carry terms between each stage. However, many more gates and wires are required, and the overall structure is much less regular than that of Figure 2. For these reasons, such an approach will not be investigated further.

#### Carry-Select

A carry-select (or conditional carry) adder is shown in Figure 3. The carry output of the first  $M$ -bit ripple adder is used to select the proper output of the next pair of ripple adders, each with complementary carry inputs. All ripple adders operate at the same time, so overall delay consists of an  $M$ -bit ripple add



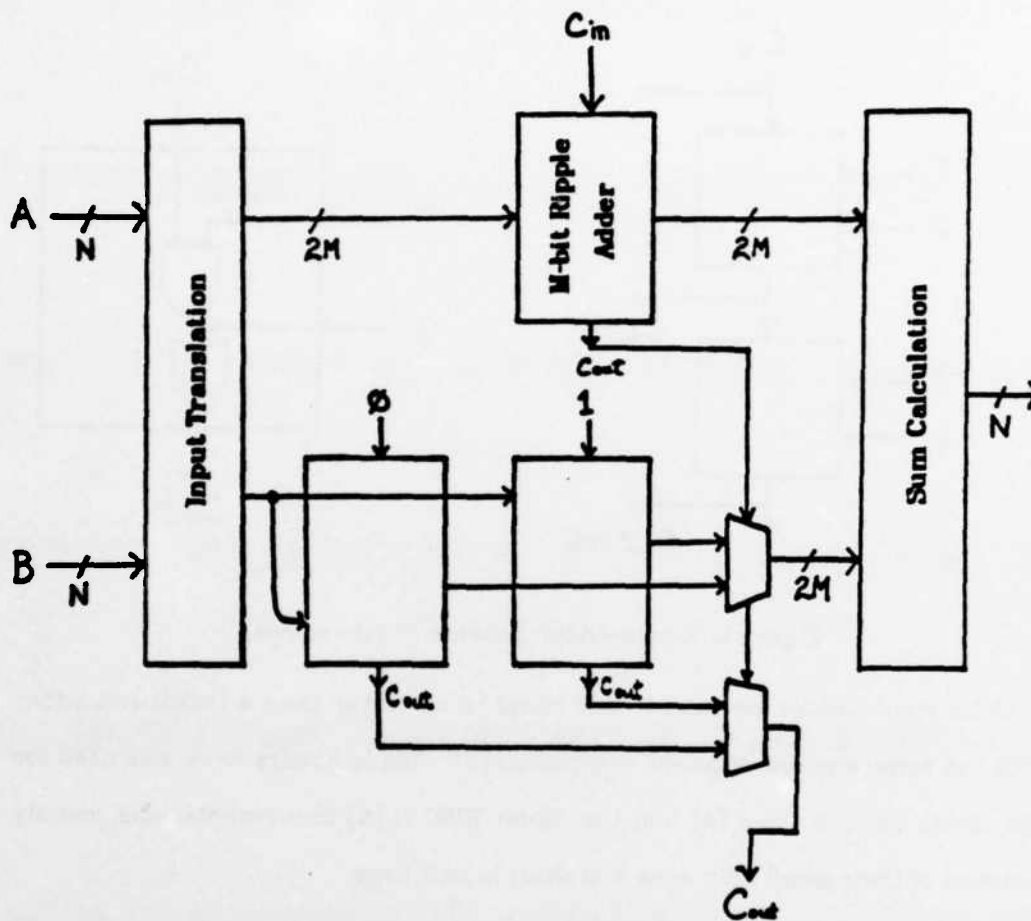


Figure 3: Carry-Select Adder

followed by a cascade of multiplexors. Carry delay goes as  $(M + \frac{N}{M})$  so there exists an optimal  $M$  yielding a lower bound in time. This choice of  $M$  for highest performance is  $\sqrt{N}$ , assuming bit delay is equal to multiplex delay.

The carry select adder is fully modular. Layout may be done with a few basic cells. No irregular wiring is required among the modules. This is important in reducing the design time, layout area, and the probability of design errors in the random wiring. Gate count (and therefore power and area) for carry calculation is nearly twice that of the ripple carry approach.

## Carry Lookahead

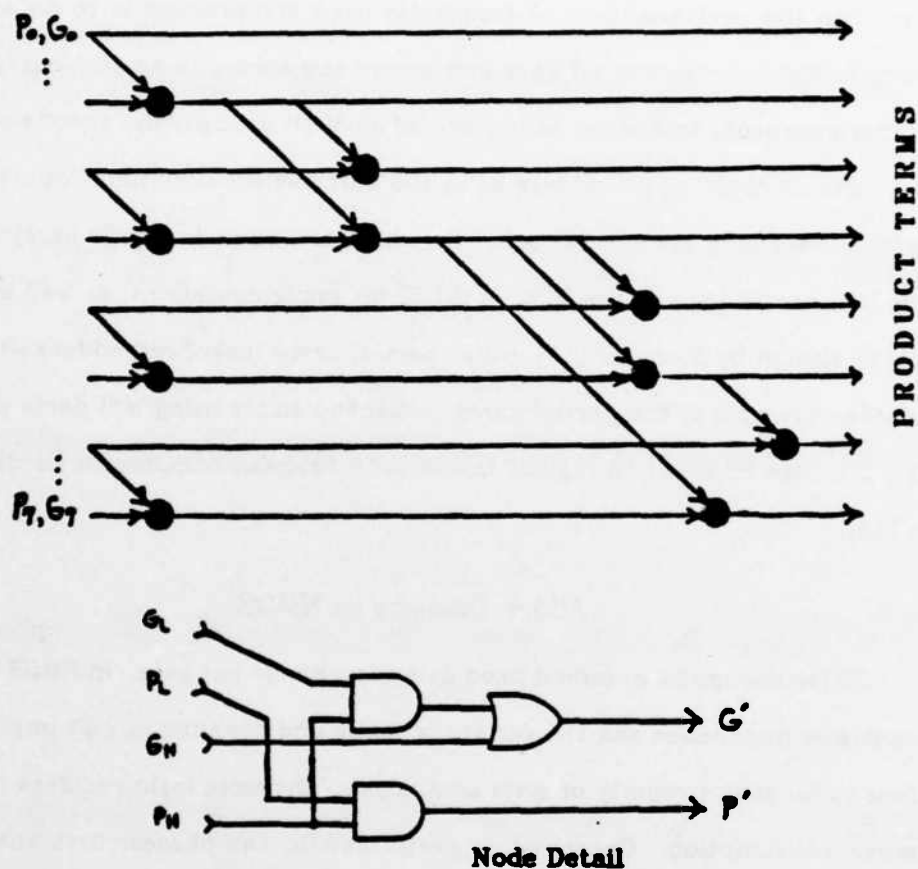


Figure 4: Parallel Adder (8 bits shown)

A full-lookahead (or parallel) adder performs calculation of all P and G product terms. Figure 4 details the organization of an 8-bit parallel adder as well as the design of the individual circuit modules. The overall delay for an N-bit parallel adder goes as  $\log_f N$  assuming a gate fan-in of  $f$ . This  $\log_f$  behavior is important in reducing carry delay for large adders. Parallel adders have been implemented in the HP Focus [8], MIPS [7] and Xerox Dragon [8] 32-bit microprocessors.

Such an approach requires nearly four times the gate count of the ripple

scheme for carry calculation. The associated increase in power consumption and irregular wiring makes this design much more costly for VLSI implementation than the previous ones. A frequently used compromise is to do a *partial* carry lookahead, trading off gate and power requirements against carry delay. In this approach, lookahead is performed in M-bit groups, the results of which are input to M-bit ripple adders as in the carry-select scheme. Results of this partial lookahead are partial products which are input to ripple carry adders. The Bellmac-32 [9] and the RISC I [10] 32-bit implementations, as well as a prototype design by Siemens [11] utilize partial carry lookahead adders with  $M=4$ . Another example is the partial carry lookahead adder using MSI parts shown in TI's TTL Data Book [1]. A regular layout for lookahead computation is discussed in [12].

### Adder Designs in NMOS

So far our model assumed fixed delay and power per gate. In NMOS the high transistor impedance and the variety of static and dynamic circuit implementations reduces the validity of such an analysis. Dynamic logic requires no static power consumption. Operation is performed in two-phases: first the output nodes are dynamically precharged, then they are selectively discharged. This selective discharge of precharged nodes without static pullups requires no ratioing of the transistors as for static logic. This allows transistors driving critical paths to be freely increased in size, to attain desired speed.

The equivalent gate model for a precharged ripple carry chain is shown in Figure 5(a). This is similar to the ripple logic in Figure 2. The NMOS dynamic ripple circuitry is given in Figure 5(b). During  $\phi_1$  the output logic levels are precharged. At this time the carry input is not allowed to discharge the chain. On  $\phi_2$  the carry input is enabled to propagate along the ripple stages by selective discharge. The G terms are also enabled to discharge the carry line.

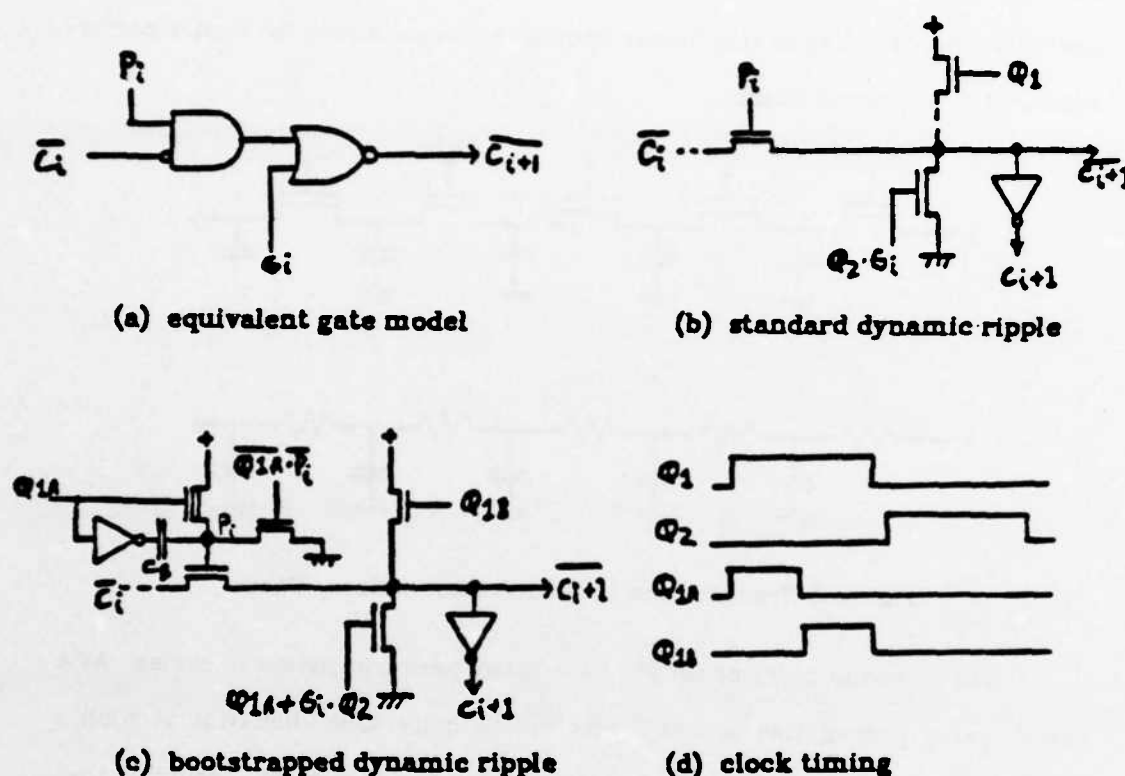


Figure 5: Dynamic Ripple Carry Circuitry

Sensing of the carry bit is performed by a static inverter; its output drives the sum logic.

A modified ripple stage is shown in Figure 5(c). It utilizes bootstrap capacitor  $C_B$  to increase conductance of the pass transistor  $P_i$ . The first clock phase  $\phi_1$  is divided into two sub-phases  $\phi_{1a}$  and  $\phi_{1b}$  as shown in 5(d). During the first sub-phase, the entire carry line is discharged to ground. At this time the bootstrap capacitor  $C_B$  is precharged. During  $\phi_{1b}$  the bootstrapping takes place, raising the pass transistor gate to 7 or 8 volts. This results in charge being coupled onto the carry line. Since it was initially discharged, this coupling does not cause it to overshoot the supply voltage. Otherwise, increased delay would result in order to discharge the carry line to a logic "0". By the end of

$\phi_1$  the carry line is precharged; operation then proceeds with  $\phi_2$  as for the previous example. Use of this bootstrapping technique allows for higher performance of a long ripple chain.

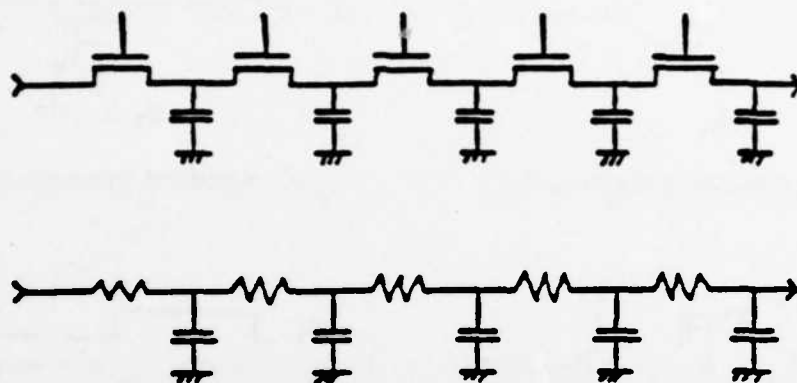


Figure 6: Transmission Line Equivalent of Carry Chain

A long dynamic carry chain will have many pass transistors in series. As a result, carry propagation across  $N$  bits will be quite slow. Behavior of such a carry chain is equivalent to that of a transmission line (Figure 6). Assuming that the pass transistors are of minimum channel length and large enough to dominate carry line parasitics, the transmission line delay becomes independent of transistor width. This is because any change in channel conductance (proportional to width) is accompanied by a proportional change in gate capacitance. The overall  $RC$  product remains constant for a particular technology.

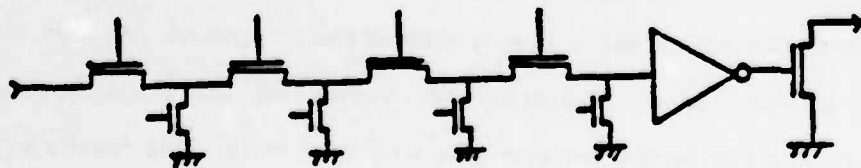


Figure 7: Carry Chain Buffering (precharge devices not shown)

One way of overcoming this long carry chain delay is to periodically buffer the carry line (Figure 7). This is equivalent to the use of repeaters on lossy transmission lines. Overall delay for long chains then becomes a linear function of carry chain length, rather than a square function as would be the case without the buffers.

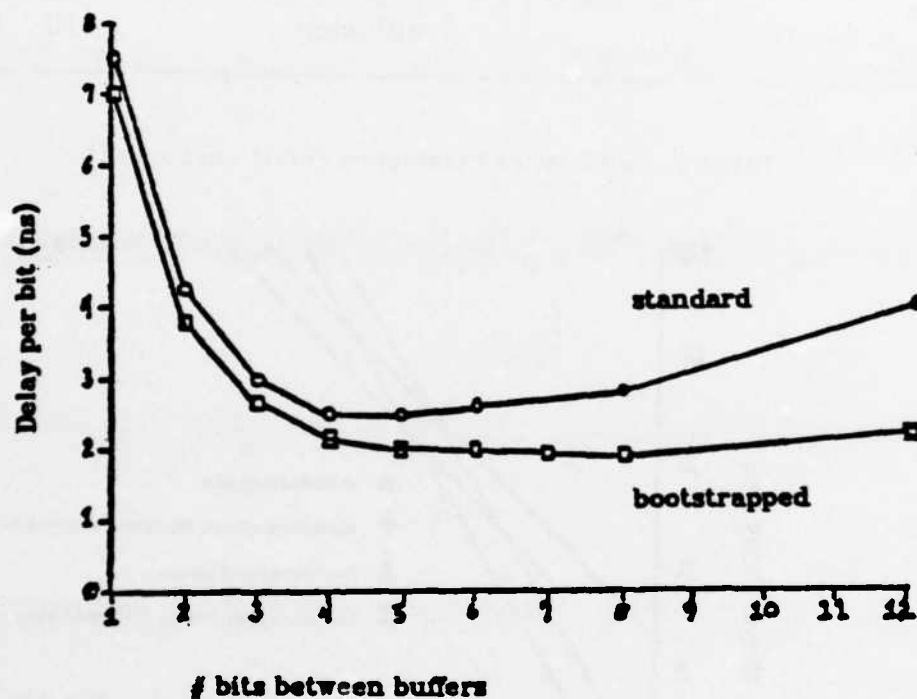


Figure 8: Buffered Carry Chain Optimization

Results of an analysis to determine optimal length of chain sections between buffers are shown in Figure 8. Data are based on SPICE simulation results using the device parameters in Table I. The ratio of parasitic to gate capacitance was 1:4. For the standard ripple carry design, four bits are optimal. This value was implemented in the Caltech OM-2 and the RISC I. The bootstrapped approach yields higher performance through eight bits, at which point only half the number of buffers are required.

$\lambda$	2.0 $\mu m$	Capacitances:	
Transistors:		metal	0.14 fF/ $\lambda^2$
$V_{ETO}$	0.9 V	diffusion bulk	0.3 fF/ $\lambda^2$
$V_{DTo}$	-3.2 V	diffusion side-wall	0.3 fF/ $\lambda$
$V_{DD}$	5.0 V	poly over field	0.2 fF/ $\lambda^2$
$V_{BB}$	-2.0 V	gate	1.6 fF/ $\lambda^2$
$\gamma$	0.75 $V^{1/2}$	gate-src overlap	0.5 fF/ $\lambda$
$k'$	20.7 $\mu A/V^2$	Resistances:	
$\mu_0$	600 $cm^2/V \cdot sec$	polysilicon	50 $\Omega/\square$
min. electr. channel L	4.0 $\mu m$	diffusion	10 $\Omega/\square$

TABLE I: NMOS Device Parameters (worst-case speed)

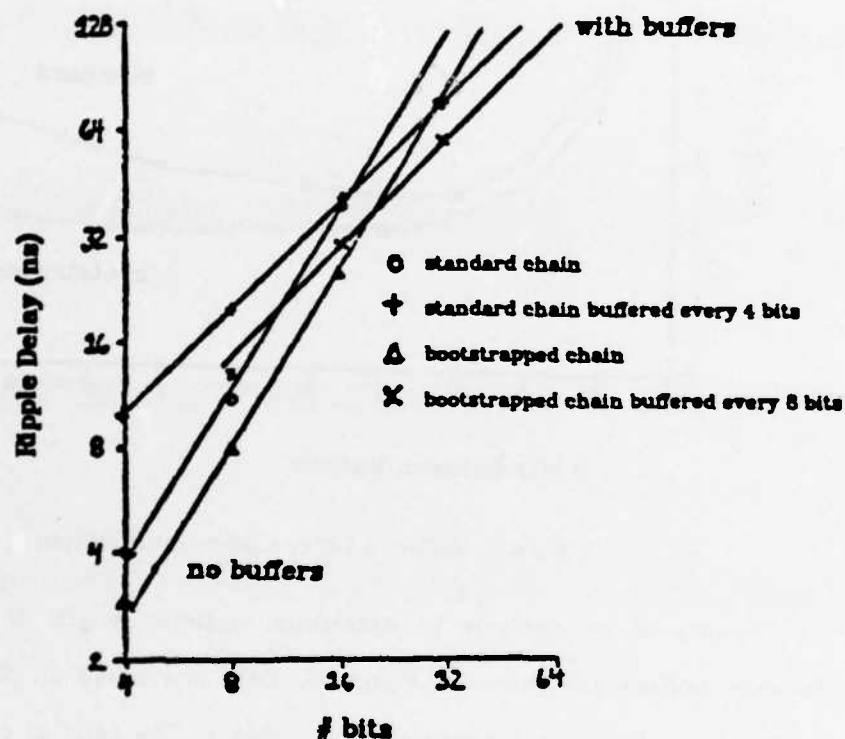


Figure 9: Comparison of Ripple Delays

A comparison of ripple carry delays with and without buffering is presented in Figure 9. Because of the added delay incurred by static buffers, the unbuffered designs are fastest for small adders. In the graph, a 16-bit adder is





the intermediate P and G product terms are required, fully dynamic logic chains are not appropriate. Delay of a series of parallel adder stages is similar to that of ripple carry stages which are buffered every bit, as both consist of a static and a dynamic gate. The result of the P and G product term calculation for all bits must then be processed in a final stage to include the carry input. Such parallel adder logic has been implemented in the MIPS and Xerox Dragon 32-bit microprocessors.

### Evaluation of Carry Schemes

Some adder strategies for representative microprocessors are summarized in Table II. It is not clear which design is best for a given datapath size, although small adders all are of the ripple type. First a comparison using the fixed gate delay model is performed, as a starting point. This is compared to results of NMOS implementation using dynamic logic and bootstrapping techniques where applicable. Although speed will be the main focus of analysis, implications on chip area and power will also be discussed.

INTEL 8080	8 Bit Ripple
MOTOROLA 68000	16 Bit Ripple
CALTECH OM-2	16 Bit Ripple
UCB RISC II	32 Bit Ripple
BELLMAC-32	32 Bit Partial Lookahead
UCB RISC I	32 Bit Partial Lookahead
HP FOCUS	32 Bit Parallel
STANFORD MIPS	32 Bit Parallel
XEROX DRAGON	32 Bit Parallel

TABLE II: Microprocessor ALU's

The fixed gate delay model has been applied to large adders in order to evaluate performance asymptotically. This is not appropriate for comparing approaches for microprocessors with adders of only 32 bits or less. This is too small for an asymptotic analysis because the constant components of delay cannot be ignored. The data presented will consider absolute delays for typical ALU sizes.

Earlier discussion of the parallel adder considered arbitrary gate fan-in. In TTL, there is little penalty for increased fan-in: most of the delay is attributed to the output driver. In VLSI, however, increased fan-in has its cost. For short paths, the gate delay is highly dependent on transistor parasitics at the output node. Increasing the number of inputs to a NOR gate adds more drain diffusion and overlap parasitics to the output node. Because fixed device ratios must be maintained for adequate noise margins in static logic, these intrinsic device parasitics cannot be eliminated from consideration.

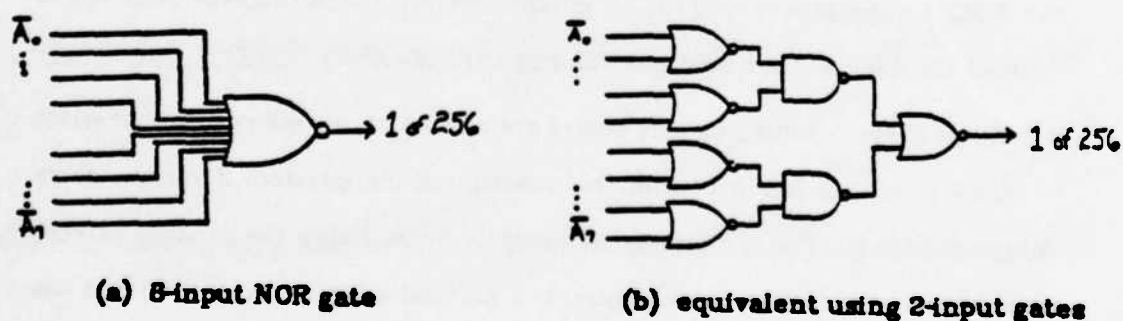


Figure 11: Realizations of 8-Input NOR Function

Delays of an 8-input, static NOR gate were compared with an equivalent realization composed of 2-input gates (Figure 11). The results, in Table III, are based on the device parameters given in Table I. Delay was measured as the time required for the output to reach 3V and 2V for logic "1" and "0", respec-

Static NOR Fan-In (K=4)	No Wire Delay		With Wire Delay	
	$t_{dLH}$	$t_{dHL}$	$t_{dLH}$	$t_{dHL}$
8 (Figure 11(a))	15ns	3ns	21ns	4ns
2 (Figure 11(b))	18ns	12ns	23ns	16ns

TABLE III: Gate Fan-In Comparison

tively. Delay to logic "0" ( $t_{dHL}$ ) is significant for the 2-input version. However, the interesting delay is that to logic "1" ( $t_{dLH}$ ): this is the limiting delay. Neglecting wire loading, the smaller fan-in incurs 20% delay penalty.

With wire delay, penalty for the 2-input constraint reduces to less than 10%. Wire loading was calculated based on a  $42 \lambda$  spacing between NOR inputs; this is the datapath pitch between each bit slice (as determined by the register file) of the RISC II microprocessor [13]. A minimum-width metal line was assumed to connect the drains of the multiple NOR input transistors.

Since these resulting circuit delays are so similar, we will restrict ourselves to circuits using a fan-in of 2 in the subsequent comparison of various carry implementations. This simplifies the analysis by reducing the number of variables to be considered. Coincidentally, the parallel adders in the MIPS and the Xerox Dragon microprocessors both use gates with a fan-in of 2 for the carry computation.

Table IV summarizes the delay and gate count for the fixed delay model carry schemes. Figure 12(a) depicts adder delay as a function of N, while 12(b) gives gate count as a first approximation of area and power requirements. Performance for the more complicated schemes is seen to improve considerably

Carry Scheme	Gate Delays for Carry	# Gates for Carry
Full Ripple	$2N$	$2N$
Carry-Select	$4\sqrt{N} - 2$	$4N - 2$
Parallel	$4\log_2 N - 2$	$8N - 3\log_2 N - 8$

TABLE IV: Carry Computation: Fixed Gate Delay Model  
(fan-in = 2, neglecting wiring delays)

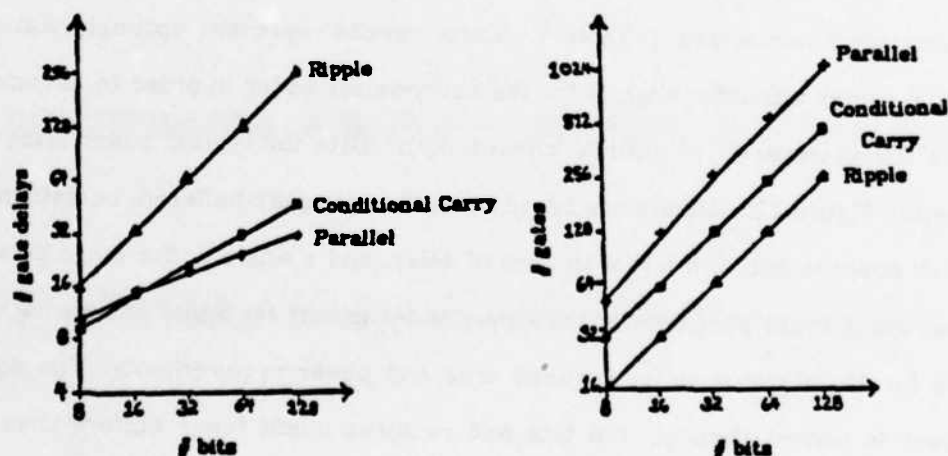


Figure 12: Delay and Gate Count Comparison

over that for the ripple scheme, though at the cost of additional area and power requirements. The carry-select design is fastest from 8 to 16 bits; it requires nearly twice the number of gates as the ripple version. The parallel design is

fastest for 32 bits and beyond, though at a gate and power cost approaching four times that of the ripple carry scheme.

Adder Scheme	Optimal Carry Delay	Inverter Count (Power)
Full Ripple	$\frac{N}{M} \tau_{chain}$	$\frac{N}{M}$
Carry-Select	$2\sqrt{\frac{N}{M} \tau_{buffer} \tau_{chain}} - \tau_{buffer}$	$\approx 2\frac{N}{M}$
Parallel	$(2\log_2 N - 1) \tau_{buffer}$	$5N - 2\log_2 N - 5$

TABLE V: MOS Implementation Comparison

(where ripple chain length  $M = \sqrt{\frac{\tau_{buffer}}{\tau_{bu}}}$ )

Results of the NMOS delay analysis using optimal size buffered ripple carry chains are summarized in Table V. These results represent optimal solutions; actual values will differ slightly for the carry-select adder in order to accommodate the granularity of optimal chain length. Data for typical adder sizes are given in Figure 13. Results are based upon the optimally buffered, bootstrapped carry chain length of 8 bits with 15ns of delay, and a single buffer stage delay of 7ns. Using these parameters, the ripple adder is best for 8 bits and also attractive for 16 bits, due to its reduced area and power requirements. The carry-select is fastest through 128 bits and requires much fewer buffers than the parallel design. In fact, for a large adder the parallel approach requires nearly 20 times as many buffers as the carry-select scheme. This high number of buffers can significantly impact power resources on the chip. Even in a CMOS implementation using no static power, the additional buffers are costly in terms of silicon area. These results differ markedly from those obtained using the

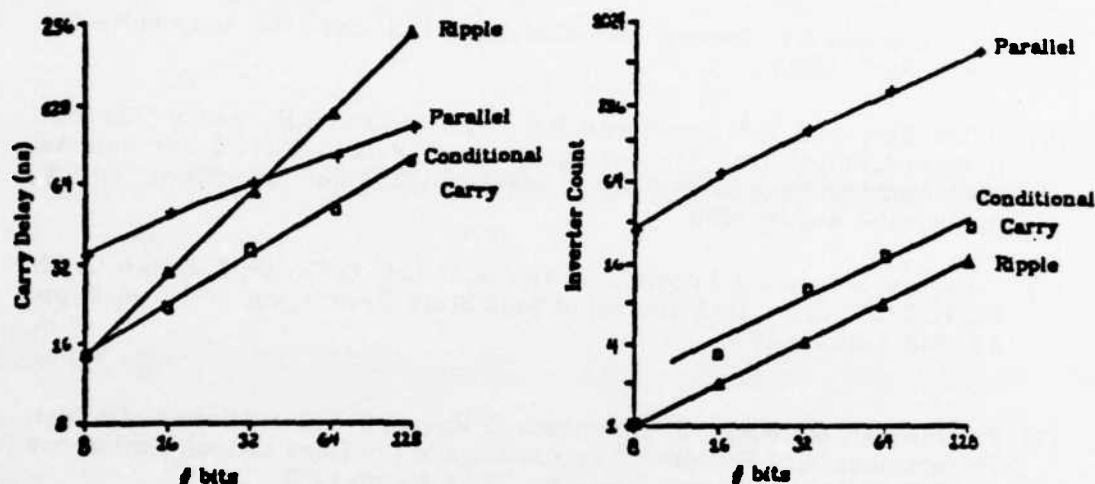


Figure 13: NMOS Carry Logic Comparison

fixed gate delay model.

A more accurate comparison must include the effect of wiring delays. The ripple adder has the shortest paths and would be least affected by such delays. The parallel adder, in contrast, has wire lengths which increase with  $N$ ; the longest connections must span half the width of the ALU. Inclusion of such delays could only lessen the gains of increased parallelism. This further reduces the applicability of the fixed gate delay analysis and makes the parallel adder unattractive for VLSI implementations.

### References

- [1] Engineering Staff of Texas Instruments, Inc., *The TTL Data Book for Design Engineers*, Application Data, SN54/74182.



- [2] H.C. Lai and S. Muroga: "Minimum Parallel Binary Adders with NOR (NAND) Gates," *IEEE Transactions on Computers*, vol. c-28, no. 9, pp. 648-659, September 1979.
- [3] Les Crudele, Motorola, private communication, January 1981.
- [4] C.A. Mead and L.A. Conway: *Introduction to VLSI Systems*, Addison Wesley, (Menlo Park, 1980).
- [5] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture," *Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83)*, Trondheim, Norway, pp. 349-359, August 1983.
- [6] J. Beyers, L. Dohse, J. Fucetola, R. Kochis, C. Lob, G. Taylor, E. Zeller: "A 32-Bit VLSI CPU Chip," *IEEE Journal of Solid-State Circuits*, vol. sc-16, no. 5, pp. 537-542, October 1981.
- [7] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," *Proceedings of the Third Caltech Conference on VLSI*, Computer Science Press, pp. 33-54, March 1983.
- [8] C. Thacker, "The Dragon Project," Lecture given at the Computer Systems Seminar, University of California, Berkeley, October 21, 1982.
- [9] B. Murphy, L. Thomas, A. MacRae: "Twin Tubs, Domino Logic, CAD Speed Up 32-Bit Processor," *Electronics*, vol. 54, no. 20, pp. 106-111, October 6, 1981.
- [10] D. Fitzpatrick, J. Foderaro, M. Katevenis, H. Landman, D. Patterson, J. Peek, Z. Peshkess, C. Séquin, R. Sherburne, K. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," *VLSI Systems and Computations*, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," *VLSI Design*, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and *Computer Architecture News (ACM SIGARCH)*, vol. 10, no. 1, pp. 28-32, March 1982.
- [11] M. Pomper, W. Beifuss, K. Horninger, W. Kaschte, "A 32-Bit Execution Unit in an Advanced NMOS Technology," *IEEE Journal of Solid-State Circuits*, vol. sc-17, no. 3, pp. 533-538, June 1982.
- [12] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, vol. c-31, no.3, pp. 260-264, March 1982.
- [13] R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, C.H. Séquin, "Datapath Design for RISC," *Proceedings of the Conference on Advanced Research in VLSI*, Massachusetts Institute of Technology, pp. 53-62, January 1982.

## CHAPTER 6:

# PROCESSOR PERFORMANCE

Previous chapters have discussed individual areas of design tradeoffs, one by one. In reality, there is much more interaction among these areas than has been suggested so far. In order to perform the analysis necessary to account for this interaction, an understanding of the entire processor design and its associated tradeoffs is required. Present 32-bit microprocessors include up to several hundred thousand transistors. Familiarization with every aspect of the design then can be a monumental task.

However, it is not necessary to consider processor behavior all the way down to the circuit level. Use of higher levels of abstraction allow some tradeoffs to be evaluated independently of circuit details or of the fabrication technology. This yields good results without overwhelming the designer and without burdening him with unnecessary detail. In some other cases, however, the strengths and weaknesses of a particular implementation technology will have an impact even at the architectural level. For example, the cost of implementing a particular function on the chip may vary so much among different processing technologies that it may become prohibitive in some instances.

Limited chip area and power resources make processor design optimization

a real challenge. A large local memory will reduce the amount of data I/O required during execution at the cost of chip resources otherwise available for other functions. Increasing the datapath wordsize has a similar effect. Optimal local memory capacity, discussed in Chapter 3, may be too costly to implement. Other strategies for performance improvement, such as increased swap support or processor pipelining, must then be considered.

Increased pipelining boosts processor throughput, as discussed in Chapters 2 and 4. A side effect of pipelining in the register file is higher memory area cost, due to the increase in the number of wordlines and bitlines necessary to support the concurrent operations. As a result, less local memory may be implemented in a given amount of chip area. This reduces the gain offered by pipelining in the first place. For fixed local memory capacity, the highly pipelined implementation will have slower register operations. Increased pipelining can even degrade system performance.

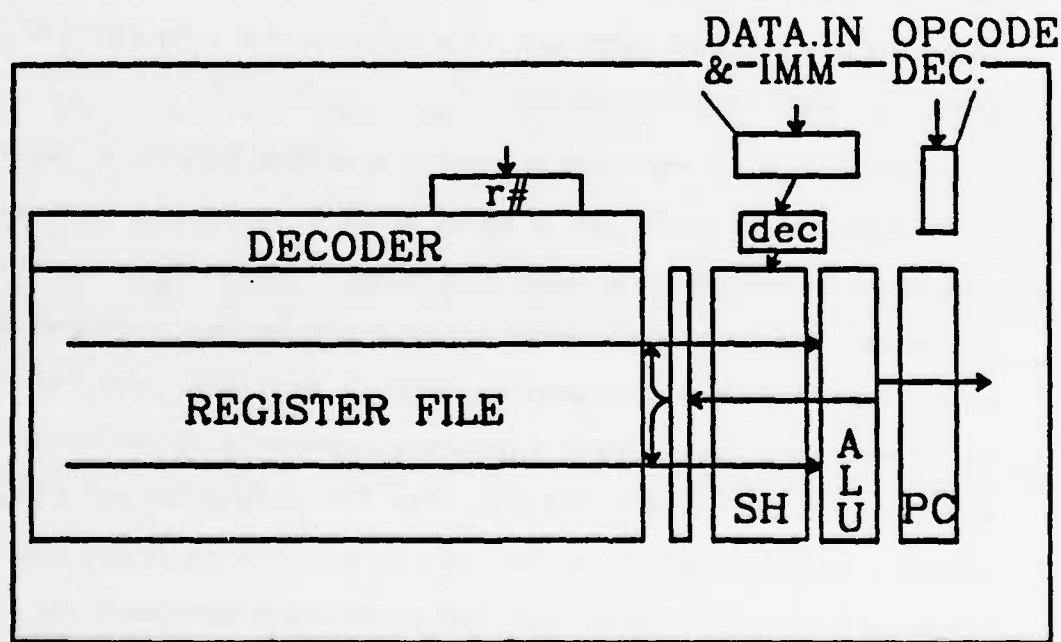


Figure 1: Chip Area Allocation in RISC II

Figure 1 shows area allocation for the RISC II microprocessor [1]. The register file occupies the majority of the datapath area; it also consumes half of the overall chip power. In contrast, the ALU occupies little area. Based on the findings of Chapter 5, it is assumed that a sufficiently fast ALU can be realized to match the register file speed. Because the register file is a limiting factor in system performance and datapath bandwidth (Chapters 3 and 4), it remains the focal point of discussion in this chapter.

### System Timing

In Chapter 4, several datapath timing schemes were evaluated. The number of sub-phases in each datapath cycle were reduced by going from a shared bitline to a dedicated bitline configuration. The delayed write and overlapped read/write schemes further reduce the cycle time. A four-fold speedup was predicted, as shown in Table I.

Datapath Timing Scheme (Chapter 4)	Bitline Configuration	Datapath Sub-Phases
Sequential	Shared Dedicated	4 $\phi$ 3 $\phi$
Delayed Write or Overlapped Read/Write	Shared Dedicated	3 $\phi$ 2 $\phi$
Delayed Write with Overlapped Read/Write	Dedicated	1 $\phi$

Table I: Datapath Timing Schemes

Overall system timing including pipelining is summarized in Table II. For all except the four-way pipeline, the datapath timing schemes vary the number of subphases, and thus system performance, by a factor of two. The overall range

of system bandwidth now doubles for an eight-fold variation. Performance of the sequential and two-way schemes is further affected by the frequency of data loads and stores; each incurs 50% or 100% overhead per cycle, respectively. For an instruction mix including 25% data loads and stores, system bandwidth can then vary ten-fold among the possible timing schemes.

Pipelining Scheme (Chapter 2)	Phases Per Instruction (Chapter 2)	Datapath Sub-Phases (Chapter 4)	Overall Cycle Time (sub-phases)
Sequential	2	4 $\phi$ 3 $\phi$ 2 $\phi$	8 $\phi$ 6 $\phi$ 4 $\phi$
Two-Way	1	4 $\phi$ 3 $\phi$ 2 $\phi$	4 $\phi$ 3 $\phi$ 2 $\phi$
Three-Way	1	4 $\phi$ 3 $\phi$ 2 $\phi$	4 $\phi$ 3 $\phi$ 2 $\phi$
Four-Way	1	1 $\phi$	1 $\phi$

Table II: System Cycle Time

### Local Memory Capacity

In Chapter 3, discussion focused on optimal local memory size. Chip area was assumed sufficient to allow its implementation. In the real world, this may not be a valid assumption. The limited chip area must be shared among many functions. System architecture and microarchitecture both play important parts in determining how much room is available for local memory. A non-ideal local memory capacity impairs performance; some improvement may be achieved with increased swap support or pipelining.

Pipelining Scheme	Bitline Configuration	Number of Bitlines	Number of Wordlines	Area Factor
Sequential	Shared	2	2	4
	Dedicated	3	4	12
Two-Way	Shared	2	2	4
	Dedicated	3	4	12
Three-Way	Shared	2	4	8
	Shared	4	2	8
	Dedicated	4	4	16
Four-Way	Dedicated	4	4	16

Table III: Bit Cell Area Variation

Increased datapath pipelining generally requires a larger bit cell (Chapter 4). An effect of pipelining, then, is a reduction in the amount of local memory which can be realized in fixed chip area. Table III presents the number of bitlines and wordlines required for various levels of pipelining. A simple estimate of relative cell size may be obtained by the product of the number of bitlines (horizontal) and wordlines (vertical) passing through the cell. In the case of a pseudostatic bitcell design, the refresh line is added to the wordline count. This model yields a four-fold variation in bit cell size. The degree of pipelining then significantly affects the amount of local memory attainable on chip.

The RISC I, with a pseudostatic, dedicated bitline cell incurs an area factor of 12. The RISC II utilizes a static, shared bitline and wordline cell; its area factor is only 4. This factor of three difference in bitcell size predicted by our simple area model closely matches actual silicon implementation (2,733.5 versus 924  $\lambda^2$  per bit).

Local memory capacity is also limited by allowable power dissipation. Power for a static register file is determined by the number of registers. For each static register, one inverter of the pair maintains a constant current to ground. There is little or no additional power required for increased pipelining. If the optimal local memory size cannot be achieved due to power limitations, then pipelining may need to be increased for higher performance.

Static bit cell power consumption (in NMOS) may be reduced by lengthening the depletion load transistors. This increases cell area. The long depletion loads in the RISC II register file lengthened the cell sufficiently to admit four bitlines without additional area penalty; however, only two were used by the two-way pipelined datapath using dedicated bitlines [2].

Power dissipation may be reduced without an area penalty, using high-resistance polysilicon loads or dynamic storage. These strategies can increase register cycle time (due to longer write and restore delays), as well as the susceptibility to soft errors induced by alpha particles.

Complementary-MOS (CMOS) is attractive due to its extremely low static power dissipation, which is determined only by leakage currents. In the past, CMOS was used primarily in specialized, low-power applications, such as in digital watches and other battery-operated products. The additional area required for "wells" or "tubs" needed to accommodate the complementary devices made CMOS too expensive to compete with the (then simpler) NMOS process. The resulting emphasis placed on NMOS technology further widened the gap in performance between these two technologies. At present, however, NMOS chips have reached their limit in allowable power dissipation. A great deal of attention is now being focused on CMOS process development; it is emerging as the primary candidate for exploiting higher device densities.



### Datapath Bandwidth

Datapath bandwidth has been discussed in terms of phases, assuming fixed phase length. Processor cycle times using this assumption were presented in Table II. In Chapter 3, register cycle time was considered to grow with the square-root of local memory capacity. Since the register delay makes up most of the cycle time, processor bandwidth decreases with enlarged memory capacity. System performance was reevaluated to include this effect.

Depending on the bitline configuration and level of pipelining employed, bit cell area was shown to vary (Table III). A larger cell requires longer bitlines and/or wordlines. This leads to increased delay, which follows the square root of cell area in a manner similar to that discussed in Chapter 3.

Pipelining Scheme	Datapath Sub-Phases	Bitline Configuration	Delay Factor	Relative Cycle Time
Sequential	4 $\phi$	Shared	2	16
	3 $\phi$	Shared	2	12
	2 $\phi$	Dedicated	$\sqrt{12}$	13.8
Two-Way	4 $\phi$	Shared	2	8
	3 $\phi$	Shared	2	6
	2 $\phi$	Dedicated	$\sqrt{12}$	6.9
Three-Way	4 $\phi$	Shared	$\sqrt{8}$	11.28
	3 $\phi$	Shared	$\sqrt{8}$	8.46
	2 $\phi$	Dedicated	4	8
Four-Way	1 $\phi$	Dedicated	4	4

Table IV: Processor Cycle Times

The relative cycle time for various pipelining and datapath timing schemes is given in Table IV. Results are based on the number of sub-phases per cycle

(Table II) times the square root of the bit cell area factor (Table III). Whereas Table II predicted an eight-fold range in datapath bandwidth, the new results shows only half of this is actually achieved. (These results assume a constant memory capacity; where several bitcell entries yield the same number of datapath sub-phases, the smaller cell was chosen).

Both the RISC I and RISC II implementations utilized two levels of system pipelining. The RISC I, with its large bitcell and 3 $\phi$  datapath timing scheme, has a relative cycle time of 10.4 using our delay model. The smaller bitcell of RISC II allowed a relative cycle time of only 8.0, despite a 4 $\phi$  datapath cycle. Comparison of actual datapath bandwidth for the fabricated chips was not possible, due to design errors in the control logic of RISC I which did not allow the full datapath bandwidth to be attained.

### Data Wordsize

A four-bit microprocessor may suffice for a microwave oven controller: little memory is addressed, operations are simple, and high speed is not required. A large microprocessor will do the job more than adequately, but it will not be cost-effective. Other applications such as number crunching of massive amounts of data pertaining to seismic exploration or weather observation require much more processing power. These scientific calculations using single and double precision floating point data require 64 and 128 bit wordsizes in conjunction with high processor bandwidth. Despite this wide range in wordsize, these applications all have one thing in common: specialization. Processor wordsize is determined unambiguously by the application.

In contrast, a time-shared, High-Level Language (HLL) programming environment supports a wide variety of uses. Data wordsize distribution includes 8-bit ASCII characters through 128-bit extended precision floating point numbers. In such an application, the choice of processor wordsize introduces

some interesting tradeoffs.

A wide datapath can execute in a single cycle operations which would otherwise require several cycles. However, the wide datapath requires proportionally more area and power resources. For a design where the datapath dominates chip area, such as the RISC II, this cost is significant. The wider datapath is also slower. Local memory cycle time (Chapter 3) as well as ALU delay (Chapter 5: conditional carry scheme) both increase with the square root of wordsize. Depending on the application, then, a wider datapath may or may not offer improved performance.

Today, typical, time-shared HLL systems utilize 32-bit processors. This allows up to 4 Gigabytes of memory to be addressed, which is sufficient for most applications. Complex arithmetic operations, such as multiplication, may be best performed by a co-processor on the system bus. Such a co-processor may even handle larger word sizes than the CPU itself.

Assuming the CPU is required to work with words of 32 bits or less, it is interesting to observe the effect of reducing the datapath width to 16 bits. Bandwidth increases by 41% due to the smaller ALU and local memory size. Overall performance then will be improved if less than 29% of the instructions require double cycles for 32-bit data.

These multiple cycles include 32-bit data loads and stores as well as ALU and shift operations. Additionally, each program branch (jump, call, return) may modify the upper half of the 32-bit address. This also requires an additional cycle. Extra cycles due to normal program counter incrementing may be neglected, since they are very infrequent.

An additional incentive for reducing wordsize is that more functionality can be added using the resources made available. Increased local memory capacity, better swap support, and more specialized ALU functions (such as 2-bit multiply)

can be added to further increase performance.

In practice, the "dead time" between clock phases as well as the clock delays themselves reduce the 16-bit speed improvement. Also, more registers will be utilized in the 16-bit processor, since each 32-bit datum requires two registers. Although this may not justify increased window size since only five or less registers are typically used per procedure [3], swapping overhead will increase for the partial swap scheme. A more detailed examination of data lengths for the particular application is necessary in order to evaluate the impact of reduced wordsize.

### Designing for Limited Chip Resources

As we have seen, the local memory occupies the largest part of the datapath area on the chip and it is the most critical component determining processor bandwidth. In designing for limited area, realizable local memory capacity is reduced with pipelining. Local memory capacity may also be limited by maximum die size in one dimension, which sets a limit on the overall length of the datapath. In Figure 1, the RISC II local memory size was restricted in the number of registers by the maximum mask pattern size and package cavity. For that design, the critical chip cost due to pipelining is attributed to the number of wordlines.

Table V compares area and length costs per unit bandwidth for fixed capacity local memory. These costs are given in terms of area- and length-delay products in order to account for processor bandwidth variation; they are obtained by multiplying the cycle time (Table IV) by the area or length factor (Table III). The highest performance return for a given amount of area or length is seen to occur for the two-way pipelined, delayed-write scheme with shared bitlines. This is similar to the approach used in the RISC II microprocessor [4].

Pipelining Scheme	Datapath Sub-Phases	Bitline Configuration	Area-Delay Product	Length-Delay Product
Sequential	4 $\phi$	Shared	64	32
	3 $\phi$	Shared	48	24
	2 $\phi$	Dedicated	166	55.4
Two-Way	4 $\phi$	Shared	32	16
	3 $\phi$	Shared	24	12
	2 $\phi$	Dedicated	83	27.7
Three-Way	4 $\phi$	Shared	90	23
	3 $\phi$	Shared	68	17
	2 $\phi$	Dedicated	128	32
Four-Way	1 $\phi$	Dedicated	64	16

Table V: Relative Chip Area and Length Costs per Unit Bandwidth  
(costs given as Area- and Length-Delay products to reflect performance)

Figure 2 presents the "Tower of Hanoi" benchmark data for the fixed-swap scheme from Chapter 3. Performance is compared among the pipelining schemes of Table V, using the best implementation for each level of pipelining. Relative performance is plotted as a function of chip area, in units of the area factor times the number of local memory windows. Since one window in RISC is reserved for global data, entries begin at two windows.

In accordance with the area-delay product in Table V, the 2-way pipelined implementation (#2 in the figure) yields the best performance with little chip area. It even outperforms the three-way version (#3) over the entire range. The four-way version (#4) is not better until three or four times the area is available; maximum performance improvement over the two-way implementation is about 50%.

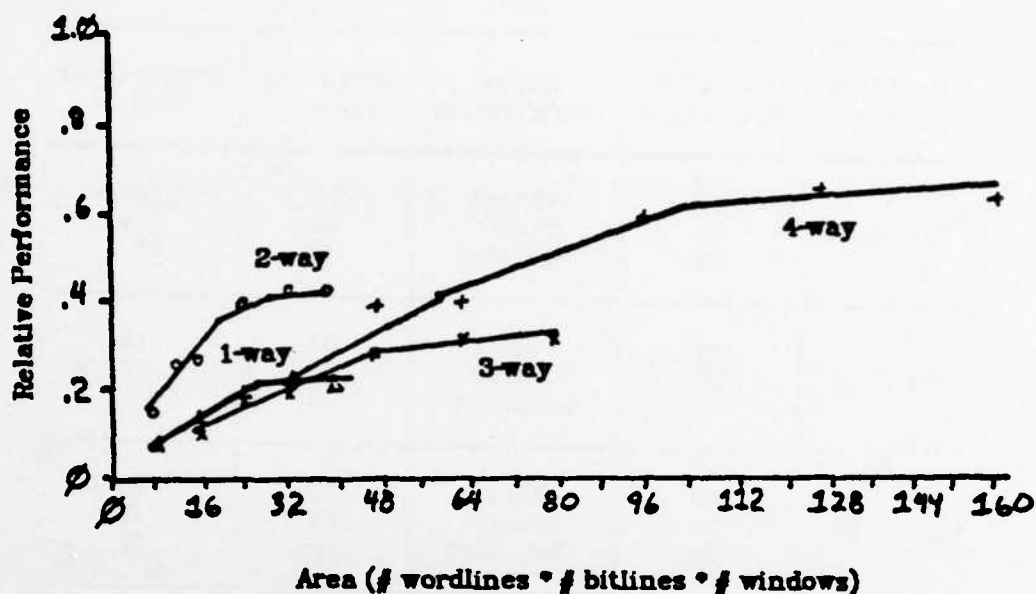


Figure 2: Pipelined System Performance Versus Local Memory Area  
(RISC II executing "Tower of Hanoi," fixed swaps, 2 cycles per register)

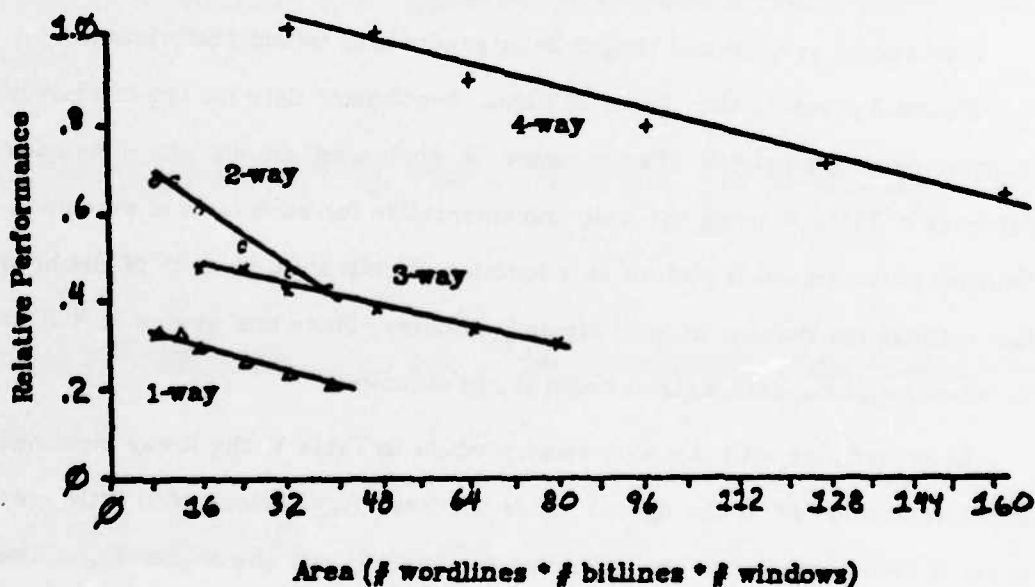


Figure 3: Pipelined System Performance Versus Memory Area  
(RISC II executing "Tower of Hanoi," partial swaps)

Figure 3 presents data using the more efficient partial swap method of local memory management, which was seen to perform the best in Chapter 3. With such a reduction in swap overhead, performance now degrades noticeably as the local memory capacity is increased, due to the increase in register cycle time. Overall performance improves by 50% versus the fixed swap scheme, while requiring only two windows (less than a third of the capacity required for the fixed swap scheme) for peak performance. As before, the four-way version yields 50% better throughput than the two-way, at a cost of three times the register area. The two-way pipeline remains superior to the three-way pipeline.

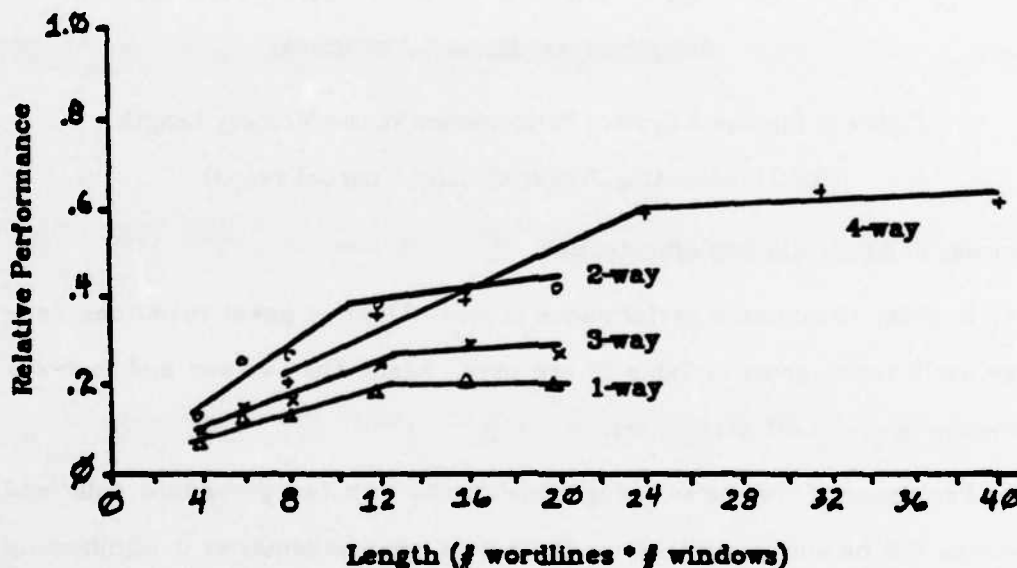


Figure 4: Pipelined System Performance Versus Memory Length  
(RISC II executing "Tower of Hanoi," fixed swaps, two cycles per register)

Figures 4 and 5 present similar results, this time in terms of the chip length constraint. Performance is given versus the number of bit cell wordlines times the number of windows. Relative performance of the pipelined schemes is similar to that in Figures 2 and 3. However, the variation in length cost is not as dramatic as that for area; only a factor of two in length separates the optimal



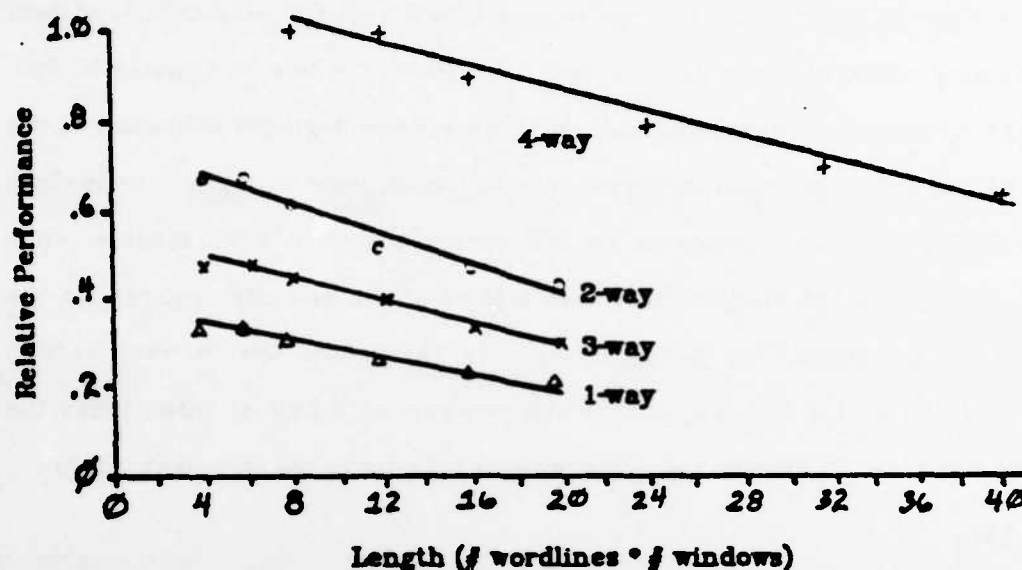


Figure 5: Pipelined System Performance Versus Memory Length  
(RISC II executing "Tower of Hanoi," partial swaps)

two-way and four-way implementations.

In order to compare performance in view of limited power resources, relative cycle times given in Table IV are used. Again, the two-way and four-way schemes are the best performers.

Performance measured using benchmarks with few procedure calls and returns will be similar to that for the partial swap scheme, as it significantly reduces the swap cost. In this case, optimal local memory size consists of only a couple windows.

Overall, the four-way pipeline gives the best performance, as expected. However, limited chip size may not allow this performance to be attained. In this case, the two-way pipeline may offer the best results. These results will vary with the amount of data I/O cycles and data dependencies encountered by the system.

As we have seen, processor design optimization in VLSI is a complex task, which must account for the limited resources available on a chip. The microarchitect must not only be familiar with the limitations of the integrated circuit technology available; he must also have some knowledge of the demands of the programming environment for which the processor is designed. This is truly a great challenge for the microarchitect.

### References

- [1] M.G.H. Katevenis, R.W. Sherburne, D.A. Patterson and C.H. Séquin: "The RISC II Micro-Architecture", Proceedings of the IFIP TC10/WG10.5 International Conference on Very Large Scale Integration (VLSI '83), Trondheim, Norway, pp. 349-359, August 1983.
- [2] R.W. Sherburne, M.G.H. Katevenis, D.A. Patterson, C.H. Séquin: "Datapath Design for RISC," Proceedings of the Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, pp. 53-62, January 1982.
- [3] D. Halbert, P. Kessler: "Windows of Overlapping Register Frames," CS 292R Final Class Report, Computer Science Division, University of California, Berkeley, Spring 1980.
- [4] D.T. Fitzpatrick, J.K. Foderaro, M.G.H. Katevenis, H.A. Landman, D.A. Patterson, J.B. Peek, Z. Peshkess, C.H. Séquin, R.W. Sherburne, K.S. VanDyke: "VLSI Implementations of a Reduced Instruction Set Computer," VLSI Systems and Computations, Carnegie-Mellon University Conference, Computer Science Press, pp. 327-336, October 1981. Also in: "A RISCy Approach to VLSI," VLSI Design, vol. II, no. 4, pp. 14-20, 4th qtr. 1981, and Computer Architecture News (ACM SIGARCH), vol. 10, no. 1, pp. 28-32, March 1982.

## CHAPTER 7:

# CONCLUSIONS

There are many tradeoffs to be considered in the design of a microprocessor. Often, these tradeoffs are interrelated and thus increase the difficulty of the task of the chip designer. In order to simplify understanding of these issues, this work has first present individual design areas in which tradeoffs can be made. Each of these design areas has been discussed individually in order to clarify the range of choices and their associated costs. Later, overall chip design was viewed with reference to all of these design tradeoffs combined in different ways.

In Chapter 1 the special constraints of VLSI single-chip processors were introduced. The high cost of custom design favors a simple and regular implementation. The RISC architecture addresses these issues by simplifying the instruction set and thereby reducing the control logic on the chip. This not only frees up valuable chip area, but also reduces design time significantly. The notion of limited chip resources (area, pins, power) sets the context for the rest of the paper. Attention is focused on the datapath itself, since it dominates chip area in RISC implementations, and its performance limits overall system speed.

System pipelining was investigated in Chapter 2. With careful design of the

datapath, pipelining may produce significant performance gains. As the degree to which pipelining is exploited is increased, however, data and jump dependencies make it more difficult to attain further speedup. As a result, a careful study of program behavior is necessary in order to accurately assess the value of various levels of pipelining. At some point, limited chip resources are better utilized to speed up other critical paths in the system rather than to support added pipelining.

Local memory tradeoffs were discussed in Chapter 3. A fundamental limit to performance exists due to the memory I/O traffic alone. Data memory traffic can be significantly reduced through the use of an on-chip local memory organized in multiple banks. Careful study is necessary in order to determine the ideal size of this local memory. A large local memory reduces datapath bandwidth and consumes resources available for other functions; too small a local memory will result in a processor that is restricted by data I/O. In some cases, however, more sophisticated hardware support for local memory management can compensate for this performance loss. Local memory design was a critical factor in optimizing the performance of the RISC microprocessors.

Datapath timing for register-based machines was examined in Chapter 4. Several schemes were presented in order to reduce the number of required clock phases in each datapath cycle. The corresponding increase in concurrency requires different register bitcell designs. In some cases, additional circuitry is needed in order to eliminate data dependencies within the datapath itself.

Design tradeoffs for the ALU were discussed in Chapter 5. Several adder schemes were compared: ripple, carry-select, and parallel. An initial analysis was performed based on the assumption of fixed gate delay, which is applicable to TTL-based implementations. Next, results of NMOS circuit simulations were

utilized to obtain a more realistic comparison of these schemes for VLSI implementation. Because dynamic logic and bootstrap techniques are available in NMOS technology, these results differ significantly from those obtained with the fixed gate delay model. The NMOS ripple carry performed best at 8 bits, while the carry-select was optimal through 128 bits. The parallel adder was determined to be undesirable for VLSI implementation because of its large area and power consumption. This contrasts with the TTL-based results, where the parallel adder is most attractive.

In Chapter 6, all of the previous design areas were considered together in order to evaluate overall processor performance under the constraint of limited chip resources. Higher levels of pipelining were found to be of diminishing return; the bit cells needed to support increased concurrency reduce the bandwidth of the datapath. The two-way pipelined system with a register file using shared bitlines and a delayed write scheme was found to make the most efficient use of limited local memory area. Such a design was utilized in the RISC II microprocessor.

Each system application requires its own analysis for optimization of performance. Additionally, design decisions must be continually reassessed as the available chip resources and constraints change with improvements in technology. It is hoped that the ideas brought out in this paper will address the nature of critical processor design tradeoffs and will prove useful to other designers faced with the task of fitting a high-performance processor onto a single VLSI chip.

# LOCAL MEMORY IN RISCS

*Robert W. Sherburne Jr., Manolis G.H. Katevenis,  
David A. Patterson, and Carlo H. Séquin*

Computer Science Division  
Department of Electrical Engineering and Computer Sciences,  
University of California, Berkeley, CA 94720

## ABSTRACT

Performance of the RISC II microprocessor is investigated as a function of local memory size. A larger local memory effectively reduces data I/O traffic arising from procedure calls and returns. Datapath bandwidth, however, is also reduced due to the increased register cycle time. A conflict then exists between the desire for maximum datapath bandwidth, and the need to reduce data I/O overhead.

Since the local memory occupies a significant portion of the chip, it is important to ensure effective usage of limited silicon resources. Programming environments which include many nested procedures benefit significantly from a multiple-bank local memory scheme. On the other hand, those with few procedures may suffer from the increased register cycle time. In addition to the programming environment, the register-bank swapping strategy, bank overflow interrupt overhead, and swap I/O support affect optimal memory size.

## Introduction

Traditionally, decisions in computer design have been based in two different camps: architecture and implementation. Architecture concerns the design as it is reflected in the instruction set: instruction and data word sizes, data types, and addressing modes. Implementation concerns the microarchitecture: system partitioning, placement, communication and timing; and, at a lower level, circuit design and device technology. Low levels of integration have permitted a wide semantic gap to exist between the architect and the chip designer: the limited repertoire of SSI/MSI chips offered little input for architectural influences.

Enter LSI and VLSI, and we suddenly find ourselves confronted with the prospect of integrating an entire system on a few chips. Each chip can now take on architectural per-

sonality. Thus, the architecture-implementation semantic gap is reduced. What this means is that more direct interdependencies exist between the two levels. The microarchitect must now evaluate architectural decisions in the context of limited chip resources.

Performance of a given architecture is limited by memory I/O and datapath cycle times. From the architect's point of view performance is I/O limited, and datapath delay can be neglected. Conversely, the designer considers the datapath bandwidth to be the performance limit, with memory I/O neglectable. Thus, two solutions may exist for design of a particular system. The goal of this paper is to examine these solutions in the context of the local memory scheme used in RISC II.

## Ideal Performance of RISC II

The RISC II is the second in a series of 32-bit, NMOS microprocessors developed at U.C. Berkeley [1]. It runs at 500ns per cycle, using an 8Mhz clock rate, and was fully functional in its first silicon run. The RISC instruction set consists solely of single register-to-register operations [2]. This simple and regular implementation reduces control complexity, chip area, and design time while supporting pipelined execution [3]. The simple RISC instruction set is a better match to highly optimizing compilers than complex instruction sets [4]. Such optimization can also reduce dependency overhead inherent in pipelined implementations [5], allowing more effective use of available datapath bandwidth.

A drawback of such an instruction set is the high I/O bandwidth required for external memory. In order to reduce this penalty, the RISC microarchitecture includes support for subroutine call and return, one of the most time-consuming operations in typical high-level



language programs [6]. Register saves and restores are reduced by employing a multiple-bank register file, or local memory. The banks are organized as stack levels, so that register save or restore need be performed only during stack overflows or underflows.

RISC II includes eight register banks, or windows, one of which is reserved for interrupt processing. At any one time there are ten registers local to the present procedure level. Additionally, there are six each high and low registers which overlap adjacent procedure levels; these are used primarily for parameter passing. Each window swap (for save or restore) involves sixteen registers: the ten locals and one set of overlaps. Ten global registers are accessible from any procedure level, forming a total of 32 addressable registers.

Table I shows the relative performance of two C programs as the number of windows on the chip varies. These results are based on earlier studies of procedure behavior and register file management overhead for RISC[7,8]. Both benchmarks, Tower and Puzzle, nest to a depth of twenty. However, Tower has significantly more procedure calls (19% versus 0.7% for Puzzle), and makes much more intensive use of the multiple windows. Performance in such a programming environment improves considerably through the use of, say, seven windows. Puzzle, on the other hand, performs well with only two windows.

WINDOWS	1	2	3	5	7	9	=
TOWER							
19% Dynamic Call & Return	7.08	3.02	2.52	1.38	1.10	1.02	1.00
PUZZLE							
0.7% Dynamic Call & Return	1.17	1.02	1.00	1.00	1.00	1.00	1.00

TABLE I: Normalized RISC II Execution Time  
(relative to case of infinite windows)

Typical programs have a procedure call or return every twenty instructions, so the benchmarks shown here represent extremes [9]. In consideration of limited chip resources, it may be attractive to tailor the local memory size to specific environments. This allows resources to be freed for performance improvement in other areas.

### Cost of Fixed-Size Window Swaps

The cost of register window overflow is determined by two factors: the overhead of servicing an interrupt, and the data I/O bandwidth. The RISC II microprocessor incurs a penalty of about thirty instructions for the window overflow/underflow interrupt routine, in addition to the register swap cost. The single I/O bus implementation supports one I/O operation per cycle, meaning that each Load or Store takes two cycles. Although each window swap is costly, there is a sufficient number of windows on chip so that the swaps are few. Reducing the local memory size degrades performance significantly due to this high cost of swapping. With fewer windows, better swap interrupt and I/O support is important.

At compile time, the dynamic procedure profile is not known. Therefore the compiler cannot anticipate window overflows. For this reason, overflows must be detected on chip. It is the cost of handling this interrupt which accounts for thirty RISC II instructions. The compiler can, however, anticipate swaps for a single window implementation: every executed call or return would require a save or restore, respectively.

Since each swap utilizes the same protocol (sixteen adjacent registers swapped to/from the current window) better data I/O support can be easily obtained. For example, a single instruction may provide all the necessary information for the swap. Then it is not necessary to fetch individual Load or Store instructions for each of the registers. Two data words may be passed on the bus each machine cycle, instead of one data word every two cycles interleaved with instruction fetching.

Table II presents RISC II performance as a function of data I/O support and local memory size. The cost of each swap includes the thirty cycles for interrupt overhead, as well as the sixteen data word transfers. Since swap overhead for Puzzle is small, only Tower is considered here. As before, seven or more windows are desirable for high performance, regardless of the level of data I/O support. This is because the interrupt overhead is so high. An exception is the single window case, which is seen to provide best performance for implementations with fewer than five windows.



WINDOWS	1	2	3	5	7	9	$\infty$
One-Half Data I/O Per Cycle	7.08	4.91	3.95	1.74	1.19	1.05	1.00
Single Data I/O Per Cycle	4.04	3.90	3.19	1.55	1.14	1.03	1.00
Dual Data I/O Per Cycle	2.52	3.39	2.81	1.46	1.12	1.03	1.00

**TABLE II: Tower Execution Time with Varying Swap Support**  
(includes interrupt overhead for multiple window cases)

### Improved Swapping Strategies

Thus far, we have discussed fixed-size swap overhead of RISC II. This scheme is attractive due to its simplicity, as well as its amenability to better I/O support. However, such a scheme swaps all registers in the window, whether they were used or not. A study of several C programs has determined that a mean of four registers are used per procedure in RISC [7]. Therefore, the fixed-swap scheme performs four times the number of save and restore data I/O actually necessary.

In order to keep track of which registers have been used, a "dirty bit" scheme may be implemented. During each register write, a bit is set to indicate a potential swap candidate. Swaps would vary in length, depending on the number of bits set. The increased hardware complexity necessary to support such an approach, however, is undesirable.

A single window implementation does not require this hardware. The compiler can insert code before each call which saves utilized registers. Restoring registers after a return can be done on demand. Since not all registers may need to be restored, further reduction in I/O is anticipated. Save overhead may be eliminated by performing a data memory write in parallel with all register file writes. This requires a dual-bus microarchitecture which can perform an instruction and data access each cycle, such as the MIPS microprocessor [10]. Overall swap overhead may then reduce by more than a factor of eight.

Although we have described alternative strategies for local memory management, we have not yet addressed effective use of on-chip memory area. These alternatives reduce off-chip register save space by a factor of four, but on-chip memory still attains only 25% utilization. A variable size window scheme addresses

this issue by allowing dynamic allocation of window size. Several register banks can be implemented for efficient replacement strategy, and procedures may span bank boundaries. This scheme requires additional hardware, in the form of pointers for each procedure domain, and an adder to calculate the physical addresses of the registers.

Performance comparison of these schemes is presented in Table III. All cases assume single data I/O per cycle support and four registers per procedure. Interrupt overhead is included in all multiple window, as well as the variable size, implementations. With few windows, significant performance improvement is observed. Using more efficient swapping strategies, high performance may be attained with fewer windows.

WINDOWS	1	2	3	5	7	9	$\infty$
Full-Bank Register Swaps	4.04	3.90	3.19	1.55	1.14	1.03	1.00
Partial Register Swaps	1.76	2.16	1.62	1.41	1.10	1.02	1.00
Partial Swaps with Write Save	1.38	1.58	1.31	1.21	1.05	1.01	1.00
Variable Size with Half Bank Swaps	3.91	1.60	1.25	-	-	-	1.00

**TABLE III: Tower Execution Time for Various Swap Schemes**  
(one data I/O per cycle assumed for all cases)

### Register File Delay

Up to this point, only I/O limited performance has been discussed. From the designer's point of view, attention should also be focused on datapath bandwidth. Especially for RISCs, where each execution cycle consists of a standard register-to-register operation, the datapath cycle time determines maximum system performance.

The machine cycle of the RISC II consists of a dual-port register read, followed by an ALU or shift operation; these latter operations overlap register write and bitline precharge of the previous instruction. The machine cycle is then limited directly by the register file read-write-precharge cycle time.

Increased local memory size is accompanied by proportionally greater bitline loading. To reduce the bitline discharge delay, the wordline transistors may be widened at some cost in addressing delay. Optimal wordline transistor size yields a register file cycle delay which goes as the square root of memory size [11]. Datapath bandwidth can then be traded off for reduced I/O with a larger local memory.

Table IV includes the effect of variable register cycle delay. The partial register swap schemes using a single window yield the best performance. A single window, fixed-swap implementation with dual data I/O per cycle approaches the performance of the seven window RISC II with half data I/O per cycle. Execution time for Puzzle, with little swap overhead, follows the register cycle time dependence with memory size; it executes nearly twice as fast with one window as it does with seven.

WINDOWS		1	2	3	5	7	9
Full-Bank	$\frac{1}{2}$	7.08	6.00	5.57	3.01	2.38	2.35
Register Swaps with Varying Data I/O per cycle	1	4.04	4.78	4.50	2.68	2.28	2.31
	2	2.52	4.14	3.96	2.53	2.24	2.31
Partial Register Swaps		1.78	2.84	2.29	2.44	2.20	2.28
Partial Swap with Write Save		1.38	1.93	1.85	2.09	2.10	2.28
Variable Size with Half Bank Swaps		3.91	1.95	1.76	-	-	-
Normalized RISC II Register Cycle Time (ignoring swaps)		1.00	1.22	1.41	1.73	2.00	2.24

TABLE IV: Datapath Bandwidth Limited Execution Time (smaller local memory is faster; using Tower benchmark)

### Conclusions

Chip design tradeoffs in VLSI must be made using both architectural and circuit design considerations. One measure of local memory cost, delay, yields two minimum execution time scenarios: I/O limited, and datapath bandwidth limited. Because of the limited number of pads which can be placed on a chip, memory I/O is a severe bottleneck in system performance. For this reason, a large local memory was chosen

for RISC II. Presently, memory speed is increasing, making datapath bandwidth a more critical determinant of system performance. In the future, increased chip resources will support greater local memory hierarchy [12]; the I/O bottleneck may then be replaced by datapath limited performance.

### References

- [1] M. Katevenis, R. Sherburne, D. Patterson, C. Séquin: "The RISC II Micro-Architecture," presented at the VLSI 83 Conference, Trondheim, Norway, August 1983.
- [2] D. Patterson & C. Séquin, "A VLSI RISC," IEEE Computer, vol. 15, no. 9, pp. 8-21, September 1982.
- [3] J. D. Wright: "Relation of Microcode to Future Machine Design," COMPCON Digest of Papers, March 1983, pp. 104-106.
- [4] G. Radin, "The 801 Minicomputer," Proc. ACM SIGARCH, pp. 39-47, March 1982.
- [5] T. Gross: "Code Optimization Techniques for Pipelined Architectures," COMPCON Digest of Papers, March 1983, pp. 278-285.
- [6] D. Patterson & C. Séquin, "RISC I: A Reduced Instruction Set VLSI Computer," Proc. ACM SIGARCH, pp. 443-457, May 1981.
- [7] D. Halbert & P. Kessler, "Windows of Overlapping Register Frames," CS 292R Final Reports, UC Berkeley, Spring 1980.
- [8] Y. Tamir & C. Séquin: "Strategies for Managing the Register File in RISC," to be published, IEEE Trans. on Computers, 1983.
- [9] D. Ditzel & H. McLellan: "Register Allocation for Free: The C Machine Stack Cache," Proc. ACM SIGARCH, pp. 48-56, March 1982.
- [10] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross: "Design of a High Performance VLSI Processor," Third Caltech Conference on Very Large Scale Integration, Computer Science Press (Rockville, 1983), pp. 33-54.
- [11] R. Sherburne, "Processor Design Tradeoffs in VLSI," Ph.D. Dissertation (in preparation), UC Berkeley.
- [12] D. Patterson & C. Séquin: "Design Considerations for Single-Chip Computers of the Future," IEEE Journal of Solid-State Circuits, vol. SC-15, no. 1, February 1980, pp. 44-52.

# Strategies for Managing the Register File in RISC

YUVAL TAMIR, STUDENT MEMBER, IEEE, AND CARLO H. SÉQUIN, FELLOW, IEEE

**Abstract**—The RISC (reduced instruction set computer) architecture attempts to achieve high performance without resorting to complex instructions and irregular pipelining schemes. One of the novel features of this architecture is a large register file which is used to minimize the overhead involved in procedure calls and returns. This paper investigates several strategies for managing this register file. The costs of practical strategies are compared with a lower bound on this management overhead, obtained from a theoretical *optimal strategy*, for several register file sizes.

While the results concern specifically the RISC processor recently built at U.C. Berkeley, they are generally applicable to other processors with multiple register banks.

**Index Terms**—Cache fetch strategies, computer architecture, procedure calls, register file management, RISC, VLSI processor.

## I. INTRODUCTION

INVESTIGATIONS of the use of high-level languages show that procedure call/return is the most time-consuming operation in typical high-level language programs [8], [9] due to the related overhead of passing parameters and saving and restoring of registers. The RISC architecture [8], [9] includes a novel scheme that results in highly efficient execution of this operation.

In conventional, register-oriented computers, the procedure call/return mechanism is based on a LIFO stack of variable size *invocation frames* (activation records). When a procedure is called, an area on top of the stack is used for storing the input arguments, saving the return address and register values, allocating local variables and temporaries, and, if the procedure calls another procedure, storing output arguments. A procedure's invocation frame denotes this area on the stack. At any point in time, the number of invocation frames in the stack is the current *nesting depth*. The invocation frame of the calling procedure overlaps that of the called procedure so that the memory locations containing the parameters passed from the calling procedure to the called procedure are part of both frames.

In most computers, register/register operations can be performed faster than the corresponding memory/memory operations. Therefore, the most heavily used local variables and temporaries are placed in registers. When a procedure is called, it must save the value of all the registers it will use and restore these values before returning control to the calling

procedure. Analysis of the dynamic behavior of Pascal and C programs, executing on a VAX 11/780, has shown [8], [9] that saving and restoring register values and writing and reading of parameters from the common area of the caller and the callee are responsible for more than 40 percent of the data memory references.

In RISC, the call/return mechanism is based on *two* LIFO stacks. One of the stacks (henceforth "STACK1") contains *fixed size* frames which hold scalar quantities of the invocation frame (i.e., scalar input arguments, the return address, scalar output parameters, and scalar local variables and temporaries). The second stack (henceforth "STACK2") contains variable size frames, some of which may be empty (i.e., their size is zero). This stack is used for all nonscalar variables which are normally placed on the single stack in conventional computers. It is also used for scalars in case there is not enough space in the fixed size frame on STACK1.

The size of the STACK1 frame in RISC was determined based on a study by Halbert and Kessler [5]. The dynamic behavior of nine noninteractive UNIX™ C programs was analyzed. These programs included the main part of the C compiler *ccom*, the Pascal interpreter *pi*, the UNIX copy command *cp*, the *troff* text formatter, and the UNIX *sort* program. This study showed that a fixed frame size of 22 "words" (22 registers), with an overlap of six "words" between adjacent frames, is sufficient for all the scalar variables and arguments in over 95 percent of the procedure calls.

The implementation of STACK2 in RISC is identical to the implementation of the single LIFO stack in conventional computers: the stack itself resides in memory, there is a processor register that serves as a stack pointer, and there is another register that serves as the frame pointer [4]. There is no special hardware support for operations on STACK2 but, due to STACK1, such operations are far less frequent than operations on the LIFO stack of conventional computers. Since the implementation and operation of STACK2 is identical to those of the stack in conventional computers, STACK2 will not be discussed any further in this paper.

In conventional computers, registers are used for storing part of the invocation frame of the currently executing procedure (i.e., the top frame on the stack). In RISC, there is a large register file that is divided into several fixed size "register banks," each of which can hold one STACK1 frame. Since each STACK1 frame partially overlaps the previous STACK1 frame and the next STACK1 frame, each register bank shares

Manuscript received July 14, 1982; revised January 3, 1983. This work was supported by the Defense Advanced Research Projects Agency under ARPA Order 3803, and monitored by Naval Electronic System Command under Contract N00039-81-K-0251.

The authors are with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

™ UNIX is a trademark of Bell Laboratories.

some of its registers with the two neighboring register banks.

The STACK1 frame used by the currently executing procedure, is always in one of the register banks. At each point in time, the contents of one of the register banks are addressable as registers, thus providing a "window" into the register file. This register bank is always the one containing the STACK1 frame of the currently executing procedure. A procedure call modifies a hardware pointer and "moves" the window to the next register bank in the register file, where the STACK1 frame of the called procedure resides. Thus, for example, register 15 (R15) in the calling procedure is in a different physical position in the register file from R15 in the called procedure, although the operand specifier for R15 is identical in the two procedures.

A return instruction restores the previous value of the above mentioned hardware pointer so the previous values of all the registers are "restored" without any data movement. Furthermore, no memory references are required for passing arguments since they are passed in registers which are in the region of overlap between the register banks containing the STACK1 frames of the caller and the callee.

By using this scheme, a procedure call in RISC can be made as fast as a jump and with fewer accesses to data memory than are required in conventional computers.

Since the size of the register file is limited, there is a need for a mechanism which will handle the case when the procedure nesting depth exceeds the number of STACK1 frames which fit in the register file. When a procedure call is executed, a new "empty" register bank is needed. If all the register banks in the register file are in use, an "overflow" occurs. This overflow causes a trap which is handled by operating system software. The operating system must free one or more register banks to make room for the new frame. Since the STACK1 frames in the register banks which are "freed" must be preserved, the software copies the frames to a conventional LIFO stack which is kept in memory and contains only STACK1 frames.

When a return instruction is executed, the window must be moved to a register bank containing the previous frame (i.e., the frame of the calling procedure). If all the register banks are free (i.e., the calling frame is not resident), an "underflow" occurs. This underflow causes a trap, upon which the operating system software loads one or more frames from memory where they were stored when an overflow occurred.

The register file is simply a write-back cache of STACK1. The cache blocks are the STACK1 frames. The top few frames of STACK1 are in the register file while the rest are in memory. When an underflow occurs, one or more occupied STACK1 frames are *fetched* from memory. When an overflow occurs, one or more register banks are "freed." This can be interpreted as "fetching" empty STACK1 frames from memory. Since in both cases the "fetching" is done by software, there is great flexibility in defining the cache *fetch strategy* (algorithm) [10]. This strategy determines the number of frames to be moved to/from memory when an overflow/underflow occurs.

In this paper, several fetch strategies are considered. A theoretical "optimal strategy" is developed and is used as a

reference point for evaluating the performance of several practical strategies. In addition, the effect of register file size on the performance of different strategies is investigated.

## II. THE OPTIMAL STRATEGY

In this section an *optimal strategy* for managing the register file will be discussed. This strategy requires unbounded look-ahead (possibly to the end of the call/return trace) and is therefore only useful as a lower bound on the cost of practical strategies. A proof that the proposed strategy is, in fact, "optimal" is presented.

### A. Definitions

In order to facilitate further discussion, some formal definitions are required.

When a program is executing, its nesting depth constantly changes: every procedure call increases the nesting depth by one and every return decreases the nesting depth by one. Hence, for every execution of a program, there is a corresponding sequence of nesting depths. This sequence will be called a *procedure nesting depth sequence* (PNDS).

**Definition 1:** A *procedure nesting depth sequence* (PNDS) is a sequence of integers  $D = (d_1, d_2, \dots, d_n)$  where  $d_1 = 1$ ,  $d_i > 0$  for  $1 \leq i \leq n$  and  $|d_i - d_{i-1}| = 1$  for  $2 \leq i \leq n$ .

The integer  $i$  is an index into the PNDS;  $d_1$  is the nesting depth at the beginning of the program. For each  $i$ ,  $2 \leq i \leq n$ ,  $d_i$  is the nesting depth after  $i - 1$  calls and returns are executed (i.e., after  $i - 1$  changes in the nesting depth). Henceforth, an index into the PNDS will be called a *location*. An example of a PNDS is shown in Fig. 1.

The frames of STACK1 are numbered from 1 to  $m$  (with  $m$  being the current nesting depth, i.e., the number of the frame of the currently executing procedure). The top (i.e., highest numbered) few frames of the stack are always in the register file while the rest are in memory.

**Definition 2:** The *register file position* (RFP) is the number of the lowest numbered frame which is in the register file.

When an overflow occurs, the lowest number frame(s) in the register file are copied to memory and the register banks they occupy in the register file are "freed." This increases the register file position. Similarly, when an underflow occurs the RFP is decreased. Thus, the number of times the RFP is changed during the execution of the program is equal to the sum of the number of overflows and the number of underflows which occur.

**Definition 3:** A *register file move* (RFM) denotes an increase or decrease in the register file position.

**Definition 4:** The *size* of the register file move is the absolute value of the difference between the RFP before the move and the RFP after the move.

If the current nesting depth is  $d$ , the STACK1 frame being used by the currently executing procedure, is the one labeled  $d$ . The register file position must be such that this frame is contained in the register file. Hence, if the register file can hold  $w$  frames and if the RFP is  $p$ , then  $p \leq d < p + w$ . Before execution begins, the RFP is some positive integer  $p_0$ . During the execution of a program with a PNDS  $D = (d_1, d_2, \dots, d_n)$ , for each nesting depth  $d_i$ , the corresponding RFP  $p_i$  must be such that the above condition is satisfied, i.e.,  $p_i \leq d_i < p_i + w$ .



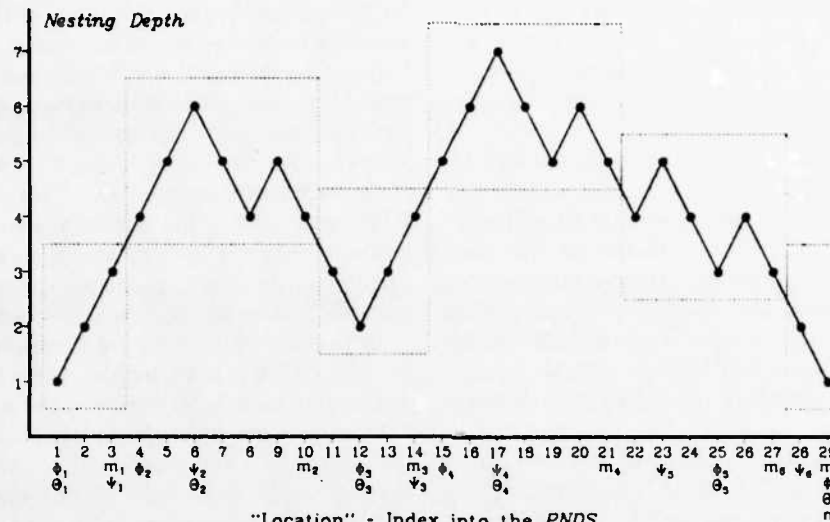


Fig. 1. PNDS and optimal RFP's.

**Definition 5:** Given a PNDS  $D = (d_1, d_2, \dots, d_n)$  and a register file that can hold  $w$  frames, a valid *register file position sequence* (RFPS) is a sequence of RFP's:  $P = (p_0, p_1, p_2, \dots, p_n)$  such that the  $p_i$ 's are positive integers and for all  $i$ ,  $1 \leq i \leq n$ ,  $p_i \leq d_i < p_i + w$ .

There is a one-to-one correspondence between nesting depths in the PNDS and RFP's in the RFPS. Successive RFP's,  $p_{j-1}$  and  $p_j$ , in the RFPS may be unequal or equal depending on whether the register file position is modified between the  $j-2$  and  $j-1$  change in the nesting depth.

**Definition 6:** If  $P = (p_0, p_1, p_2, \dots, p_n)$  is an RFPS, an RFM is said to occur in *location*  $j$  ( $1 \leq j \leq n$ ) of  $P$ , if and only if  $p_j \neq p_{j-1}$ .

The number of RFM's which occur during some interval in which the program is executing is of interest in this paper. The interval is defined as a subsequence of the RFPS (which corresponds to a subsequence of the PNDS).

**Definition 7:** If  $P = (p_0, p_1, p_2, \dots, p_n)$  is an RFPS, the number of RFM's occurring in location range  $[i, j]$  of  $P$ , where  $1 \leq i \leq j \leq n$ , is the number of unique integers  $k$ , such that  $i \leq k \leq j$  and  $p_k \neq p_{k-1}$ . This number will be denoted by  $\text{RFM}_P[i, j]$ .

**Definition 8:** If  $P = (p_0, p_1, p_2, \dots, p_n)$  is an RFPS, the *memory traffic* occurring in location range  $[i, j]$  of  $P$ , where  $1 \leq i \leq j \leq n$ , is the total number of STACK frames moved to/from memory as the RFP is set to  $p_i, p_{i+1}, \dots, p_j$  successively. This number is denoted by  $\text{MT}_P[i, j]$ :

$$\text{MT}_P[i, j] = \sum_{k=i}^j |p_k - p_{k-1}|.$$

### B. What is an "Optimal Strategy"?

There is some overhead involved in handling overflow/underflow traps: saving the current state, determining the cause of the trap, activating the proper trap handling routine, restoring state, and returning to normal execution. Hence, it is desirable to minimize the number of register file overflows and underflows. In addition, there is the direct *cost* involved in actually moving the data to/from memory. For each register

file move, this cost is proportional to the number of frames moved (i.e., to the size of the register file move). Hence, it is desirable to minimize the number of frames moved for each overflow/underflow, i.e., the memory traffic which is the result of overflows and underflows.

The problem of finding the "best" RFPS is similar to finding optimal strategies for handling page faults in virtual memory systems. In virtual memory systems it is also desirable to minimize both the number of page faults (since there is overhead involved in handling such faults) and the I/O involved in moving memory pages to/from disk or drum. For the virtual memory problem, Belady [1] developed an "optimal" page replacement algorithm which causes the fewest possible page faults for a program which executes in a fixed number of main memory page frames. Belady's algorithm is not realizable since it requires knowledge of the future portion of the page trace.

In the next sections it is shown that if the entire call return trace of the program (i.e., the PNDS) is known, there exists a RFPS which achieves *both* the minimum number of overflow/underflow traps and the minimum memory traffic resulting from register file moves. It is further shown that knowledge of the entire PNDS is *necessary* for achieving an optimum RFPS.

### C. The Existence of an Optimal RFPS

In order to prove the existence of an optimal RFPS, an algorithm for deriving such an RFPS from a given PNDS, is presented. The optimality of the RFPS produced by the algorithm is shown by proving that no other valid RFPS can have fewer register file moves or result in less memory traffic.

An optimal RFPS can be obtained as follows. We start with the RFP at 1 and keep it there until the nesting depth exceeds the number ( $w$ ) of register banks in the register file. Now the RFP must be changed, i.e., an RFM must occur. In order to determine the *optimal size* of the RFM, we must look ahead in the call/return trace (i.e., in the PNDS). Starting from the current location, we determine the longest subsequence of the PNDS for which a constant RFP is possible (i.e., in which the

difference between the maximum nesting depth and the minimum nesting depth does not exceed  $w - 1$ ). The new RFP is chosen so that it is valid for this entire subsequence. From the end of this subsequence we repeat the procedure until the entire PNDS is covered.

Special handling is required when determining the RFP for the last subsequence in the PNDS. In this case the difference between the maximum and minimum nesting depth within the subsequence may be less than  $w - 1$ . Hence, there is some freedom in setting the RFP. In order to minimize the memory traffic, the new RFP is chosen so that it is valid for the entire subsequence and the absolute value of the difference between the new RFP and the previous RFP is minimized. An example of a PNDS and the corresponding optimal RFPS is shown in Fig. 1.

More formally, the procedure can be stated as follows. Given an arbitrary PNDS  $D = (d_1, d_2, \dots, d_n)$ , an optimal RFPS  $P = (p_0, p_1, p_2, \dots, p_n)$ , for a register file that can hold  $w$  frames, can be obtained as follows:

```

[1] let  $i = 1, p_0 = 1$ 
[2] repeat
[3]   let  $E = (d_i, d_{i+1}, \dots, d_m)$ 
      where  $m$  is the maximum integer such that
       $i \leq m \leq n$  and  $\max(E) - \min(E) < w$ 
[4]   if  $(m < n)$  or  $(p_{i-1} > \min(E))$  then
[5]     for  $j = i$  to  $m$ 
[6]       let  $p_j = \min(E)$ 
[7]   else
[8]     for  $j = i$  to  $m$ 
[9]       let  $p_j = \max(E) - w + 1$ 
[10]  let  $i = m + 1$ 
[11] until  $i > n$ 

```

First, it must be shown that the algorithm generates a valid RFPS for the given PNDS. Proof of the validity of the algorithm and of the generated RFPS requires proving the following lemmas.

**Lemma 1:** The repeat and for loops terminate after a finite number of iterations, i.e., the algorithm always terminates.

**Proof:** Since  $n$  is finite and  $i$  is incremented by at least 1 during each iteration of the repeat loop,  $n$  is an upper bound on the number of iterations through the repeat loop.

It is always true that  $i \geq 1$  and  $m \leq n$ . Hence,  $n$  is an upper bound on the number of iteration through the for loop (either one) each time it is entered. ■

**Lemma 2:** For all  $i$ ,  $1 \leq i \leq n$ ,  $p_i \leq d_i < p_i + w$ , i.e., the RFP's generated by the algorithm are valid.

**Proof:** From the algorithm, if  $p_i$  is set in step 5, then  $p_i \leq d_i$  [since  $p_i = \min(E)$ ] and  $d_i - p_i < w$  (since  $\max(E) - \min(E) < w$ ). Hence,  $p_i \leq d_i < p_i + w$ .

If  $p_i$  is set in step 7, then  $p_i \geq d_i - w + 1$  (since  $p_i = \max(E) - w + 1$ ) and  $p_i + w - 1 - d_i < w$  (since  $\max(E) = p_i + w - 1$  and  $\max(E) - \min(E) < w$ ). From the first inequality,  $d_i \leq p_i + w - 1$  and from the second inequality  $p_i < d_i + 1$ . Hence,  $p_i \leq d_i < p_i + w$ . ■

The proof of the optimality of the generated RFPS requires some additional notation. The subsequence  $E$  which is defined during the  $k$ th iteration of the repeat loop will be denoted  $E_k$

(it corresponds to the  $k$ th setting of the RFP). The corresponding integer  $m$  will be denoted  $m_k$ . For convenience in notation, we define  $m_0 = 0$ . The number of iterations that the repeat loop executes before terminating will be denoted by  $K$  (it corresponds to the number of times that the RFP is adjusted). Note that  $1 \leq m_1 < m_2 < \dots < m_K = n$ .

For each location range,  $[m_{k-1} + 1, m_k]$ , the RFP's in the RFPS generated by the algorithm are constant. Within this location range,  $\Phi_k$  and  $\Psi_k$  are the locations of the first occurrence of the minimum and maximum nesting depths, respectively. More formally, see the following.

**Definition 9:**  $\Phi_k$  and  $\Psi_k$  are the smallest integers, such that for each  $k$  ( $1 \leq k \leq K$ ), both are in the location range  $[m_{k-1} + 1, m_k]$ , where  $d_{\Phi_k} = \min(E_k)$  and  $d_{\Psi_k} = \max(E_k)$ .

In order to prove the optimality of the RFPS generated by the algorithm, it must be shown that this RFPS results in the lowest possible memory traffic. This will be done by using induction on the  $K$  boundaries of  $K - 1$  location ranges. These boundaries are defined below and are denoted by  $\Theta_k$ , for all  $k$  such that  $1 \leq k \leq K$ . The boundary point  $\Theta_k$  is the location of the first minimum or maximum nesting depth within the location range  $[m_{k-1} + 1, m_k]$ . If the RFP in the RFPS generated by the algorithm for location range  $[m_{k-1} + 1, m_k]$  is less than the RFP for location range  $[m_{k-2} + 1, m_{k-1}]$ , then  $\Theta_k = \Phi_k$ , otherwise  $\Theta_k = \Psi_k$ . More formally:

**Definition 10:**  $\Theta_k$  is an integer such that for each  $k$ ,  $2 \leq k \leq K$ ,  $\Theta_k = \Phi_k$  if  $d_{\Phi_k} < d_{\Phi_{k-1}}$ , and  $\Theta_k = \Psi_k$  otherwise. For convenience in notation, we define  $\Theta_1 = 1$ .

Fig. 1 shows the PNDS from the execution of Ackerman's function with arguments (2, 1). The dotted squares show the "optimal" RFP's for a register file that can hold three frames. In this example, five RFM's are necessary ( $\text{RFM}_P[1, 29] = 5$ ,  $K = 6$ ) and the memory traffic resulting from those RFM's is 12 frames ( $\text{MT}_P[1, 29] = 12$ ).

Let  $Q = (q_0, q_1, q_2, \dots, q_n)$  be an arbitrary valid RFPS for  $D$ .

The rest of this section contains a formal proof that the number of RFM's in  $P$  and the memory traffic resulting from those RFM's are at most equal to the number of RFM's in  $Q$  and the memory traffic resulting from those RFM's, respectively.

**Lemma 3:** If  $K > 1$ , then for all  $k$ ,  $1 \leq k < K$ ,  $\text{RFM}_Q[1, m_k + 1] \geq k$ .

**Proof:** See the Appendix.

From the algorithm, for all  $k$ ,  $1 \leq k \leq K$ ,  $d_{\Psi_k} - d_{\Phi_k} \leq w - 1$ . It is now shown that for  $1 \leq k \leq K - 1$ ,  $d_{\Psi_k} - d_{\Phi_k} = w - 1$ .

**Lemma 4:** If  $K > 1$ , then for all  $k$ ,  $1 \leq k \leq K - 1$ ,  $d_{\Psi_k} - d_{\Phi_k} = w - 1$ .

**Proof:** See the Appendix.

It should be noted that Lemma 4 makes no claims about the value of  $(d_{\Psi_K} - d_{\Phi_K})$ , i.e., it makes no claims about the case  $k = K$ . From the algorithm it is clear that  $d_{\Psi_K} - d_{\Phi_K} < w$ . So it is quite possible that  $d_{\Psi_K} - d_{\Phi_K} < w - 1$ .

**Lemma 5:** If  $K > 1$ , for all  $k$ ,  $1 \leq k \leq K - 1$ , for all  $i$ ,  $m_{k-1} + 1 \leq i \leq m_k$ ,  $p_i = d_{\Phi_k} = d_{\Psi_k} - w + 1$ . For the last subsequence, i.e.,  $k = K$ : if  $\Theta_K = \Phi_K$ , then  $p_i = d_{\Phi_K}$ , else  $p_i = d_{\Psi_K} - w + 1$ .

**Proof:** See the Appendix.

**Lemma 6:** If  $K > 1$ , then for all  $k$ ,  $1 \leq k \leq K$ ,  $MT_Q[1, \Theta_k] \geq MT_P[1, \Theta_k] + |q_{\Theta_k} - p_{\Theta_k}|$ .

*Proof:* See the Appendix.

Using the above lemmas, we can formally prove the "optimality" of the RFPS generated by the algorithm.

**Theorem 1:** The RFPS  $P$  generated by the algorithm for the PNDS  $D$  is an optimal RFPS for  $D$ , i.e., if  $Q$  is an arbitrary valid RFPS for  $D$ , then

$$RFM_P[1, n] \leq RFM_Q[1, n]$$

and

$$MT_P[1, n] \leq MT_Q[1, n].$$

*Proof:* If  $K = 1$ , then there are no RFM's in  $P$  so  $RFM_P[1, n] = 0$ ,  $MT_P[1, n] = 0$ , and the theorem holds.

Assume  $K > 1$ . In the algorithm, all the RFP's, corresponding to the same subsequence, are set to the same value (step 5 or step 7). Hence, the only way that  $p_i \neq p_{i+1}$  can occur is if  $i = m_k$  for some  $k$ ,  $1 \leq k \leq K - 1$ . Thus,  $RFM_P[1, n] \leq K - 1$ .

From Lemma 3,  $K - 1 \leq RFM_Q[1, m_{K-1} + 1]$ . Since  $m_{K-1} + 1 \leq n$ ,  $RFM_Q[1, m_{K-1} + 1] \leq RFM_Q[1, n]$ . Hence,  $K - 1 \leq RFM_Q[1, n]$ . Thus,  $RFM_P[1, n] \leq RFM_Q[1, n]$ .

From Lemma 6,  $MT_Q[1, \Theta_K] \geq MT_P[1, \Theta_K] + |q_{\Theta_K} - p_{\Theta_K}|$ . Since  $|q_{\Theta_K} - p_{\Theta_K}| \geq 0$ ,  $MT_Q[1, \Theta_K] \geq MT_P[1, \Theta_K]$ . Since  $\Theta_K \leq n$ ,  $MT_Q[1, n] \geq MT_Q[1, \Theta_K]$ . Since  $d_{\Theta_K} \in E_K$  and  $d_n \in E_K$ ,  $p_{\Theta_K} = p_{\Theta_K+1} = \dots = p_n$ . Thus,  $MT_P[\Theta_K + 1, n] = 0$ , so  $MT_P[1, \Theta_K] = MT_P[1, n]$ . Hence,  $MT_P[1, n] \leq MT_Q[1, n]$ .

Q.E.D.

#### D. The Unrealizability of an Optimal Strategy

When a computer is executing a program, the entire call/return trace is not known ahead of time. In fact, it is unlikely that there is any look-ahead possible. In this section it is shown that knowledge of the entire PNDS is necessary for finding an optimal RFPS.

First, it should be noted that no simplifying assumptions about the properties of the call/return trace of "real" programs can be made. In other words, for every given sequence of integers which satisfies the definition of a PNDS (Definition 1), it is possible to construct a real program whose sequence of nesting depths is the given sequence. This is demonstrated by the program in Fig. 2 (which is written in the C language [7]). When this program is executed, its sequence of nesting depths is identical to the sequence of integers in the array *depthlist* (assuming that the sequence of integers in *depthlist* is a valid PNDS).

To show that unbounded look-ahead on the call/return trace is necessary for achieving an optimal RFPS, consider a system where there is a bounded (or nonexistent) look-ahead; more specifically, a system where at each point in time only the next  $t$  calls and returns are known in advance. (Note that in most systems  $t = 0$ .) Assume that the register file of the system can hold  $w$  frames and that there are two programs to be executed: PROG1 and PROG2. These programs have identical call/return traces for the first  $s$  calls and returns, where  $w + t < s$ . At some point, before  $s - t$  calls/returns are executed, the nesting depth (in both programs) reaches  $w + 1$ . The nesting

```
int depthlist[] = { /* This is the PNDS. 0 terminated */
    1, 2, 3, 2, 3, 4, 3, 2, 1, 0 };
int depthind = 1;
main()
{
    while (depthlist[depthind] > 1) {
        deeper(2);
        depthind = depthind + 1;
    }
    deeper(curdep);
    int curdep; /* The current nesting depth */
    {
        depthind = depthind + 1;
        while (depthlist[depthind] > curdep) {
            deeper(curdep+1);
            depthind = depthind + 1;
        }
        if (depthlist[depthind] == 0)
            exit(0);
    }
}
```

Fig. 2. A program whose "behavior" follows an arbitrary PNDS.

depth stays between 2 and  $w + 1$  until a total of  $s$  calls/returns are executed. After that, in PROG1 the nesting depth decreases and the program terminates at nesting depth 1. In PROG2, on the other hand, the nesting depth increases to  $w + 2$  and then decreases until the program terminates at nesting depth 1.

In both programs, when the nesting depth first reaches  $w + 1$ , the same information about the call/return trace is available, and therefore any strategy for managing the register file will result in the same action being taken for both programs. This action is clearly *not* optimal for at least one of the programs. For PROG1, the optimal action is to move one frame to memory. This action is not optimal for PROG2 since another overflow will occur when a nesting depth of  $w + 2$  is reached. The optimal action for PROG2 is to move two frames to memory so that only one overflow will occur during the execution of the program. Moving two frames to memory is *not* the optimal action for PROG1 since it results in unnecessary memory traffic: moving two frames to and from memory instead of one.

The fact that an optimal strategy is not realizable does not imply that all practical strategies for managing the register file are equally bad. As seen in the next section, simple changes in the strategy for managing the register file may significantly affect the cost of handling calls and returns.

### III. PRACTICAL STRATEGIES FOR MANAGING THE REGISTER FILE

In most real systems, no look-ahead at the call/return trace is possible. Thus, the decision as to how many frames should be moved to/from memory when an overflow/underflow occurs must be based on the previous behavior of the executing program or be completely independent of the PNDS of the executing program.

As indicated above, two factors contribute to the *cost* (execution time) of handling register file overflows and underflows: the handling of the interrupt/trap that is initiated by the overflow/underflow and the actual transfer of the STACK1 frames to/from memory. If the number of frames which are moved when an interrupt occurs is not fixed, some computation may be required in order to calculate this number. The cost of this calculation is included in the cost of handling the interrupt. In order to evaluate the effectiveness of different



strategies for managing the register file, these strategies can be tried out on the call/return trace of benchmark programs. The number of overflows/underflows and transfers of STACK1 frames which result from each strategy can thus be determined. These numbers can then be related to the cost of the overflow/underflow handler using the following formula:

$$\text{cost} = \alpha \times (\text{number overflows} + \text{number underflows}) + \beta \times (\text{number frames moved})$$

where  $\alpha$  and  $\beta$  are constants:  $\alpha$  is the cost of responding to the interrupt and calculating the number of frames to be moved, and  $\beta$  is the cost of moving one STACK1 frame to or from memory.

#### A. Measurement Technique

The method used for obtaining the call/return trace of the benchmark programs used in this paper relies on the fact that the call/return trace of a program executing on a RISC computer is identical to the call/return trace of the same program executing on any similar computer. In this case, the benchmark programs are all written in the C language [7], and their call/return trace is obtained from their execution on a VAX 11/780. The assembly code produced by the C compiler is processed by an editor script which inserts calls to special procedures before and after each procedure call instruction. When the program is executed, in addition to producing its normal output, it creates a file containing a string of bits. The  $i$ th bit in the string corresponds to the  $i$ th call/return executed by the program. This bit is 1 if a call was executed, 0 if a return was executed. The bit string is the call/return trace of the program. Routines which simulate different strategies for managing the register file use this string to obtain the number of overflows/underflows and the resulting memory traffic which will occur if the benchmark program is executed using the simulated strategy.

For this study, three benchmark programs were used:

<i>rcc</i>	The RISC C compiler [2] which is based on Johnson's portable C compiler [6]. The call/return trace used was generated by the compiler compiling the UNIX file concatenation utility <i>cat</i> . 88 606 calls and returns were executed and a nesting depth of 26 was reached.
<i>puzzle</i>	This is a bin-packing program which solves a three-dimensional puzzle. It was developed by Forest Baskett. During the execution of the program, 42 710 calls and returns were executed and a nesting depth of 20 was reached.
<i>tower</i>	This is a Tower of Hanoi program. The call/return trace used, was obtained for the program moving 18 disks. 1 048 574 calls and returns were executed and a nesting depth of 20 was reached.

In this paper, the cost of handling register file overflows and underflows is assumed to be directly proportional to the number of RISC instructions they require. If no calculation is needed in order to determine the number of frames to be moved, the cost of responding to the interrupt is approximately

30 instructions ( $\alpha = 30$  in the above discussion). The cost of moving one STACK1 frame is 16 instructions ( $\beta = 16$  in the above discussion).

#### B. The Cost of "Fixed" Strategies

The simplest strategy for managing the register file is to always move the same number of frames (say  $i$ ) to memory, when an overflow occurs, and always move the same number of frames (say  $j$ ) from memory, when an underflow occurs. For a register file that can hold  $w$  frames, such a strategy will be denoted *fixed*( $i, j$ ) where  $i$  and  $j$  are integers such that  $1 \leq i \leq w$  and  $1 \leq j \leq w$ .

When a *fixed* strategy is used, no computation is required in order to determine the number of frames to be moved. Hence, the equation

$$\text{cost} = 30 \times (\text{number overflows} + \text{number underflows}) + 16 \times (\text{number frames moved})$$

is used to evaluate the cost of managing the register file. This equation is also used in evaluating the cost of the optimal strategy, which serves as a lower bound on the cost of other strategies.

1) *Measurement Results*: The actual "performance" of the optimal strategy and *fixed* strategies is presented in this section. All possible fixed strategies for register files containing 3, 5, 7, 9, 13, and 17 register banks have been tried with the three benchmark programs.

Tables I-III summarize the results for each one of the three benchmark programs with six different register file sizes and for seven different strategies. The results include the number of overflows, number of underflows, memory traffic, and cost. For the optimal strategy, the "raw" numbers are presented. For the other six strategies, the figures shown are normalized with respect to the corresponding entries for the optimal strategy with the same register file size. In the three tables  $w$  denotes the number of register banks in the register file.

The *fixed* strategies included in the tables are: the best of all *fixed* strategies (i.e., the strategy resulting in the least cost) for the particular program and register file size, the worst of all *fixed* strategies (i.e., the strategy resulting in the greatest cost) for the particular program and register file size, *fixed*( $w, 1$ ) which guarantees the minimum number of overflows, *fixed*( $1, w$ ) which guarantees the minimum number of underflows, *fixed*( $1, 1$ ) which guarantees the minimum memory traffic, and *fixed*( $\lceil w/2 \rceil, \lceil w/2 \rceil$ ) which is "symmetrical."

2) *Discussion of Measurement Results*: Although the three benchmark programs used are quite different, the results show many common characteristics in their behavior, as far as the management of the register file is concerned. In addition, the results for the *fixed*( $w, 1$ ), *fixed*( $1, w$ ), and *fixed*( $1, 1$ ) strategies provide an experimental verification to the fact that the "optimal strategy," presented in Section II, does indeed minimize the number of overflows/underflows and memory traffic simultaneously.

The register file size and the way that the register file is managed can significantly affect the cost of procedure calls. Table IV shows the average number of instructions per procedure call required for managing the register file. For every

TABLE I  
FIXED STRATEGIES WITH *rcc*

Reg File Size ( <i>w</i> )	3	5	7	9	13	17
Best Fixed Strategy	(1.1)	(1.1)	(2.2)	(2.2)	(2.2)	(2.2)
Worst Fixed Strategy	(3.3)	(5.5)	(7.7)	(9.9)	(13.13)	(17.17)
Optimal Strategy (raw)	#Overflows	5828	1483	548	238	83
	#Underflows	6200	1232	474	171	71
	Mem. Traffic	19008	5554	2462	1376	458
	Cost	864936	170314	70372	34226	11948
Best Fixed Strategy (normalized)	#Overflows	1.83	1.87	1.85	1.87	1.87
	#Underflows	1.53	2.25	1.91	2.58	1.96
	Mem. Traffic	1.00	1.00	1.46	1.28	1.21
	Cost	1.31	1.50	1.59	1.80	1.44
Worst Fixed Strategy (normalized)	#Overflows	2.30	7.08	9.49	10.84	43.40
	#Underflows	2.16	8.50	10.97	14.69	50.73
	Mem. Traffic	4.24	16.85	29.33	32.86	204.46
	Cost	3.15	13.53	20.99	25.54	143.50
fixed( <i>w</i> , 1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00
	#Underflows	2.82	6.02	6.09	12.42	15.20
	Mem. Traffic	1.84	2.67	3.09	3.09	4.71
	Cost	1.89	2.96	3.61	4.05	5.81
fixed(1, <i>w</i> ) (normalized)	#Overflows	3.19	3.88	5.44	5.85	4.83
	#Underflows	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	1.96	2.07	2.40	1.94	1.75
	Cost	2.01	2.31	2.83	2.57	2.26
fixed(1, 1) (normalized)	#Overflows	1.63	1.87	2.26	2.92	2.78
	#Underflows	1.53	2.25	2.62	4.02	3.23
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00
	Cost	1.31	1.50	1.62	1.85	1.76
fixed( $\lfloor \frac{w}{2} \rfloor, \lfloor \frac{w}{2} \rfloor$ ) (normalized)	#Overflows	1.85	2.73	2.38	2.51	1.46
	#Underflows	1.55	3.29	2.76	3.46	1.70
	Mem. Traffic	2.02	4.36	4.21	4.30	3.70
	Cost	1.79	3.71	3.49	3.61	2.88

TABLE II  
FIXED STRATEGIES WITH *puzzle*

Reg File Size ( <i>w</i> )	3	5	7	9	13	17
Best Fixed Strategy	(1.1)	(1.1)	(1.1)	(2.2)	(2.2)	(3.3)
Worst Fixed Strategy	(3.3)	(5.5)	(7.7)	(9.9)	(13.13)	(17.17)
Optimal Strategy (raw)	#Overflows	738	159	28	6	1
	#Underflows	747	159	30	5	1
	Mem. Traffic	2056	514	94	30	14
	Cost	77388	17784	3184	610	284
Best Fixed Strategy (normalized)	#Overflows	1.40	1.62	1.81	1.67	1.00
	#Underflows	1.38	1.62	1.57	2.00	1.00
	Mem. Traffic	1.00	1.00	1.00	1.33	1.00
	Cost	1.22	1.33	1.36	1.53	1.00
Worst Fixed Strategy (normalized)	#Overflows	9.80	39.50	130.58	70.63	1482.00
	#Underflows	9.85	39.50	113.17	65.00	1482.00
	Mem. Traffic	21.05	122.20	505.84	255.00	2715.14
	Cost	14.54	77.79	302.62	162.59	2450.39
fixed( <i>w</i> , 1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00
	#Underflows	2.98	5.00	6.07	10.80	13.00
	Mem. Traffic	2.15	3.09	3.67	3.60	1.66
	Cost	2.05	3.04	3.79	4.38	2.94
fixed(1, <i>w</i> ) (normalized)	#Overflows	3.04	4.99	7.65	6.50	7.00
	#Underflows	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	2.16	3.09	4.34	2.80	1.00
	Cost	2.06	3.04	4.26	3.17	1.63
fixed(1, 1) (normalized)	#Overflows	1.40	1.62	1.81	2.50	7.00
	#Underflows	1.36	1.62	1.57	3.00	7.00
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00
	Cost	1.22	1.33	1.36	1.70	2.27
fixed( $\lfloor \frac{w}{2} \rfloor, \lfloor \frac{w}{2} \rfloor$ ) (normalized)	#Overflows	1.30	2.36	2.36	1.17	1.00
	#Underflows	1.26	2.36	2.36	1.40	1.00
	Mem. Traffic	1.86	4.42	5.26	2.33	1.00
	Cost	1.53	3.33	3.66	1.90	1.00

call there is a corresponding return. Hence, in this context, "procedure call" includes returning from the procedure as well as invoking it.

The data indicate that, even with the optimal strategy, the

TABLE III  
FIXED STRATEGIES WITH *tower*

Reg File Size ( <i>w</i> )	3	5	7	9	13	17
Best Fixed Strategy	(1.1)	(3.3)	(1.1)	(1.1)	(1.1)	(3.3)
Worst Fixed Strategy	(3.3)	(4.5)	(6.6)	(9.9)	(13.13)	(17.17)
Optimal Strategy (raw)	#Overflows	74898	16912	4126	1028	64
	#Underflows	74898	16912	4126	1028	64
	Mem. Traffic	262142	65534	18362	4094	254
	Cost	6666152	2063264	509792	127084	7904
Best Fixed Strategy (normalized)	#Overflows	1.75	1.11	1.98	2.00	1.98
	#Underflows	1.75	1.11	1.98	2.00	1.98
	Mem. Traffic	1.00	1.71	1.00	1.00	1.00
	Cost	1.39	1.42	1.46	1.48	1.48
Worst Fixed Strategy (normalized)	#Overflows	2.00	4.84	32.26	128.00	84.00
	#Underflows	2.00	3.88	32.26	128.00	84.00
	Mem. Traffic	3.43	10.00	97.54	577.41	419.28
	Cost	2.89	7.23	65.82	359.88	246.67
fixed( <i>w</i> , 1) (normalized)	#Overflows	1.00	1.00	1.00	1.00	1.00
	#Underflows	3.00	5.00	7.00	9.00	13.00
	Mem. Traffic	1.71	2.58	3.53	4.51	6.55
	Cost	1.86	2.79	3.76	4.75	6.77
fixed(1, <i>w</i> ) (normalized)	#Overflows	3.00	5.00	6.99	8.99	7.00
	#Underflows	1.00	1.00	1.00	1.00	1.00
	Mem. Traffic	1.71	2.58	3.52	4.50	3.53
	Cost	1.86	2.79	3.75	4.74	3.76
fixed(1, 1) (normalized)	#Overflows	1.75	1.94	1.98	2.00	1.98
	#Underflows	1.75	1.94	1.98	2.00	1.98
	Mem. Traffic	1.00	1.00	1.00	1.00	1.00
	Cost	1.39	1.46	1.46	1.48	1.48
fixed( $\lfloor \frac{w}{2} \rfloor, \lfloor \frac{w}{2} \rfloor$ ) (normalized)	#Overflows	2.33	1.11	8.47	16.48	1.00
	#Underflows	2.33	1.11	8.47	16.48	1.00
	Mem. Traffic	2.67	1.71	17.07	41.31	3.53
	Cost	2.49	1.42	12.69	29.26	2.30

TABLE IV  
COST OF REGISTER FILE MANAGEMENT PER PROCEDURE CALL

Reg File Size ( <i>w</i> )	3	5	7	9	13	17
Optimal Strategy	<i>rcc</i>	15.01	3.84	1.59	0.77	0.27
	<i>puzzle</i>	3.62	0.63	0.15	0.04	0.01
Best Fixed Strategy	<i>rcc</i>	19.73	5.77	2.53	1.23	0.39
	<i>puzzle</i>	4.43	1.11	0.20	0.08	0.01
Worst Fixed Strategy	<i>rcc</i>	47.24	52.00	33.33	19.73	17.30
	<i>puzzle</i>	52.68	64.71	45.15	6.93	32.59

cost of managing the register file may become prohibitive if the register file is too small (three register banks). In this case, for two out of the three programs (*rcc* and *tower*), it is likely that a conventional stack mechanism for handling procedure calls would have resulted in better performance. If a larger register file is used, the cost of managing the register file drops sharply. The results indicate that, for a register file of five or more register banks, this scheme compares favorably with the conventional stack mechanism.

Invoking a high-level language procedure and returning from it requires several RISC instructions in addition to those used for managing the register file. Specifically, arguments have to be copied to the area of overlap between the current STACK1 frame and the next STACK1 frame; if the procedure returns a value, it may have to be copied from this overlap area; the stack pointer and frame pointer for STACK2 may need to be updated; the actual RISC *call* and *ret* instructions must be executed. C procedures typically have less than four arguments [5]. Hence, in addition to the RISC instructions that manage the register file, between three and seven instructions will be executed for each procedure call/return pair.

If an efficient strategy (such as the "best fixed strategy") is used, the cost of managing the register file decreases as the number of register banks in the register file increases. Once this cost reaches approximately one RISC instruction per procedure call/return pair (e.g., using the "best fixed strategy" with a register file containing nine register banks), it no longer dominates the total number of instructions required for each procedure call/return. In a single chip VLSI microprocessor, chip area is a precious resource. Rather than adding more register banks (e.g., beyond nine), the limited chip area can be used more effectively for other purposes, such as an on-chip cache or hardware support for multiply, that are likely to make a greater contribution to overall processor performance. Even for the benchmarks used here, which reach a relatively high nesting depth [5], a register file with between five and nine register banks seems optimal.

Choosing a "good" strategy is critical to the success of the register file scheme. Tables II and III show that choosing the "wrong" strategy can result in more than four orders of magnitude increase in the cost of managing the register file. Furthermore, if an inefficient strategy is used, an increase in the register file size can result in an *increase* in the cost of managing the register file (since there is an opportunity to generate more useless memory traffic). In most cases, the best fixed strategy is to minimize the memory traffic (i.e., use the *fixed*(1, 1) strategy). This can be explained by the fact that the cost of moving one frame to memory and then from memory back to the register file is about the same as the cost of handling the trap when an overflow or underflow occurs. Hence, the immediate cost of unnecessarily moving a frame (which results in one frame's worth of traffic to memory and later back to the register file) is about equal to the cost of not moving a frame when it should have been moved (an extra overflow or underflow trap). In addition, if an unnecessary move is made, the cost may include the cost of an extra overflow or underflow which will occur later. Hence, the "penalty" for moving one more frame than necessary, when an overflow or underflow occurs, is greater than the "penalty" for moving one fewer frame than necessary. Thus, if the call/return sequence is random, the best fixed strategies are likely to be those that require the movement of only one or two frames when an overflow or underflow occurs. The use of such strategies is further supported by the fact that with the optimal strategy, in cases where there are more than ten overflows/underflows throughout the execution of the program, the average number of frames moved when an overflow or underflow occurs is between 1.4 and 3 and in most cases is approximately 2.

### C. Taking the Past into Account

The *fixed* strategies do not attempt to take into account the previous behavior of the executing program. It is conceivable that a strategy that does take past behavior into account would result in a lower cost, closer to that of the optimal strategy.

One way of "taking the past into account" involves keeping track of which register banks have been used since the last overflow or underflow. If two or more STACK1 frames are moved whenever an overflow or underflow occurs, it is clear that, in some cases, it will turn out that too many frames will be moved, resulting in unnecessary memory traffic. When an

overflow occurs, register banks are "freed" by copying their contents to memory. If some of the freed register banks remain unused until the next underflow, their contents remain intact and need not be copied from memory to the register file. Similarly, if too many register banks are loaded when an underflow occurs, the contents of those that are unused until the next overflow need not be copied to memory since their contents are already in the appropriate memory locations.

Many practical strategies result in unnecessary memory traffic, i.e., more memory traffic than is required by the optimal strategy. The above technique reduces the memory traffic resulting from any such strategy. Our measurements indicate that with the useless "worst fixed strategy," which produces an exorbitant number of unnecessary moves of STACK1 frames, keeping track of which register banks are used can reduce this memory traffic by up to an order of magnitude. However, with "reasonable" strategies, the gains are less impressive. If the "best fixed strategy" is *fixed*(1, 1) then clearly no gain is possible. With the *fixed*(2, 2) strategy, the decrease in memory traffic is less than ten percent. The above technique requires some extra hardware and a few more instructions in the trap handling routine. When the overhead of these extra instructions is taken into account, the total cost of managing the register file for the *fixed*(2, 2) strategy is about the same as without this extra mechanism. For the *fixed*(1, 1) strategy, the extra instructions will simply add to the cost of managing the register file without any saving in memory traffic.

We have investigated two other methods for "taking the past into account." They both involve determining the number of frames to be moved when an overflow or underflow occurs based on the previous behavior of the program. The first method (henceforth denoted C/R) is to use the call/return trace immediately preceding the overflow or underflow. The second method (henceforth denoted O/U) is to use the trace of overflows and underflows which preceded the trap being handled.

The C/R method can be implemented by adding a special shift register to the processor. Every call instruction shifts a 1 into the register and every return shifts a 0. The routine which handles the overflow/underflow trap examines the contents of this register and determines the immediately preceding call/return trace of the program. This pattern is used to access a table containing the "optimal" number of frames that should be moved, given a particular call/return pattern. This scheme adds very few instructions to the cost of handling the overflow/underflow trap.

The O/U method does not require any additional hardware. The "overflow/underflow trace" is kept in a fixed memory location and is updated each time an overflow or underflow occurs by the routine that handles these traps. The pattern in this memory location is used in the same way as the contents of the shift register for the C/R method.

Both the C/R method and O/U method require finding a mapping between "call/return patterns" or "overflow/underflow patterns" and "number of frames to be moved" so that the total cost is reduced. In order to find such a mapping (for either one of the methods) we tabulated the optimal number of frames to be moved (which can be found given unbounded look-ahead) following various call/return or overflow/un-

derflow patterns for the three benchmark programs. We attempted to use these tables to determine which patterns indicate that a single frame should be moved and in which cases moving more than one frame would be preferable. However, we could not find a single mapping which worked better than the *fixed*(1, 1) strategy for all three programs!

For the three benchmark programs used in this work, it appears that the optimal number of frames to be moved is, for all practical purposes, independent of the immediately preceding call/return pattern of length ten or less. The O/U method shows more promise but the results are inconclusive. Following a suggestion by Denning [3], we tested an O/U method which involved moving two frames after two consecutive overflows or underflows and moving one frame otherwise. For register file sizes of interest (between five and nine frames), the cost of managing the register file using this method was compared to the cost using the *fixed*(1, 1) strategy. Reductions of up to 28 percent in the number of overflows and underflows and increases of up to 59 percent in the memory traffic were measured. When the extra instructions in the trap handling routines are taken into account, the overall cost was either equal to or greater than the cost of the *fixed*(1, 1) strategy in all but one case.

#### IV. CONCLUSIONS

The success of the RISC architecture is due, in part, to the reduction in the number of memory accesses which is possible through the use of the register file [11]. We have shown that the effectiveness of the register file is dependent on choosing the "right" size for the register file and an efficient strategy for deciding how many frames should be moved to/from memory when an overflow/underflow occurs.

Our measurements indicate that with the simple *fixed* strategy, *fixed*(1, 1), the cost of managing the register file is within a factor of two of the cost of the optimal strategy (which requires unbounded look-ahead). For a register file containing more than eight register banks, the *fixed*(2, 2) strategy yields slightly better performance.

If a "reasonable" strategy is used, the cost of managing the register file is inversely proportional to its size. If the register file is too small, the number of overflows and underflows becomes prohibitively large. Since the STACK1 frames have a fixed size, the large number of overflows and underflows results in a lot of memory traffic even when the number of registers actually used (for arguments and local variables) is small. Hence, if the register file is too small, the overall cost of procedure calls may be greater than if a conventional stack mechanism is used. Our measurements indicate that if the register file contains five or more frames, the use of the register file scheme rather than a conventional stack mechanism is worthwhile.

We have attempted to use past behavior of the program in order to predict the future behavior and reduce the cost of managing the register file. So far, our attempts have not succeeded.

The first method (keeping track of which register banks have been used since the last overflow or underflow), reduces the cost of managing the register file only for inefficient strategies.

For efficient strategies, such as *fixed*(1, 1) or *fixed*(2, 2), the extra overhead in the trap handling routine was greater than the savings from the reduced memory traffic.

The two other methods attempt to determine the "optimal" number of frames to be moved from the immediately preceding pattern of calls/returns or overflows/underflows. These methods appear ineffective since we could not find a single mapping between either type of patterns and number frames to be moved, which reduces the cost for all three programs. These results, while preliminary, raise serious doubts that a mapping which reduces the cost of managing the register file for a majority of programs could be found. Even in this context, the simplest solution appears to also be the best.

#### APPENDIX

##### PROOF OF LEMMAS 3-6

*Lemma 3:* If  $K > 1$ , then for all  $k$ ,  $1 \leq k < K$ ,

$$\text{RFM}_Q[1, m_k + 1] \geq k.$$

*Proof:* By induction on  $k$ .

*Basis:*  $k = 1$ . It is shown that  $\text{RFM}_Q[1, m_1 + 1] \geq 1$ . From the algorithm,

$$\max(E_1) - \min(E_1) < w$$

while

$$\max(E_1 \cup \{d_{m_1+1}\}) - \min(E_1 \cup \{d_{m_1+1}\}) \geq w.$$

Hence, either

$$d_{m_1+1} < \min(E_1)$$

or

$$d_{m_1+1} > \max(E_1).$$

By Definition 1,  $d_1 = 1$  and  $d_i \geq 1$  for all  $i$ ,  $1 \leq i \leq n$ . Hence,

$$d_1 = \min(E_1 \cup \{d_{m_1+1}\}) = \min(E_1)$$

and

$$d_{m_1+1} = \max(E_1 \cup \{d_{m_1+1}\}).$$

Thus,

$$d_{m_1+1} - d_1 \geq w.$$

Since  $Q$  is a valid RFPS for  $D$ ,  $q_1 \leq d_1 < q_1 + w$  and  $q_{m_1+1} \leq d_{m_1+1} < q_{m_1+1} + w$ .  $d_{m_1+1} \geq d_1 + w$  and  $d_1 \geq q_1$  imply that  $d_{m_1+1} \geq q_1 + w$ . But  $q_{m_1+1} + w > d_{m_1+1}$ . Hence,

$$q_{m_1+1} + w \geq q_1 + w,$$

i.e.,  $q_{m_1+1} > q_1$ . The fact that  $q_{m_1+1} \neq q_1$  implies that at least one RFM occurs in the location range  $[2, m_1 + 1]$ , so  $\text{RFM}_Q[1, m_1 + 1] \geq 1$ .

*Induction Step:* Assuming that this lemma holds for  $k = \alpha - 1$ , where  $2 \leq \alpha < K$ , it is now proven that it holds for  $k = \alpha$ . In other words, assuming  $\text{RFM}_Q[1, m_{\alpha-1} + 1] \geq \alpha - 1$ , it is proven that  $\text{RFM}_Q[1, m_\alpha + 1] \geq \alpha$ .

If  $\text{RFM}_Q[1, m_{\alpha-1} + 1] \geq \alpha - 1$ , then either

$$\text{RFM}_Q[1, m_{\alpha-1} + 1] \geq \alpha$$



or

$$\text{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1.$$

The former case implies that  $\text{RFM}_Q[1, m_{\alpha} + 1] \geq \alpha$  (since  $m_{\alpha} + 1 \geq m_{\alpha-1} + 1$ ) and the lemma is proved. Hence, we can assume  $\text{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1$ .

The rest of the proof is similar to the proof of the *basis*:  
From the algorithm,

$$\max(E_{\alpha}) - \min(E_{\alpha}) < w$$

while

$$\max(E_{\alpha} \cup \{d_{m_{\alpha}+1}\}) - \min(E_{\alpha} \cup \{d_{m_{\alpha}+1}\}) \geq w.$$

Hence, either  $d_{m_{\alpha}+1} < \min(E_{\alpha})$  or  $d_{m_{\alpha}+1} > \max(E_{\alpha})$ .

Assume  $d_{m_{\alpha}+1} < \min(E_{\alpha})$ :

From the algorithm and the definition of  $\Psi$ ,  $d_{\Psi_{\alpha}} - d_{m_{\alpha}+1} \geq w$ . Since  $Q$  is a valid RFPS for  $D$ ,

$$q_{\Psi_{\alpha}} \leq d_{\Psi_{\alpha}} < q_{\Psi_{\alpha}} + w$$

and

$$q_{m_{\alpha}+1} \leq d_{m_{\alpha}+1} < q_{m_{\alpha}+1} + w.$$

Hence,

$$q_{\Psi_{\alpha}} + w > d_{\Psi_{\alpha}} \geq d_{m_{\alpha}+1} + w \geq q_{m_{\alpha}+1} + w.$$

i.e.,  $q_{\Psi_{\alpha}} > q_{m_{\alpha}+1}$ .

Assume  $d_{m_{\alpha}+1} > \max(E_{\alpha})$ :

From the algorithm and the definition of  $\Phi$ ,

$$d_{m_{\alpha}+1} - d_{\Phi_{\alpha}} \geq w.$$

Since  $Q$  is a valid RFPS for  $D$ ,

$$q_{\Phi_{\alpha}} \leq d_{\Phi_{\alpha}} < q_{\Phi_{\alpha}} + w$$

and

$$q_{m_{\alpha}+1} \leq d_{m_{\alpha}+1} < q_{m_{\alpha}+1} + w.$$

Hence,

$$q_{m_{\alpha}+1} + w > d_{m_{\alpha}+1} \geq d_{\Phi_{\alpha}} + w \geq q_{\Phi_{\alpha}} + w.$$

i.e.,  $q_{m_{\alpha}+1} > q_{\Phi_{\alpha}}$ .

The fact that  $q_{\Psi_{\alpha}} \neq q_{m_{\alpha}+1}$  ( $q_{\Phi_{\alpha}} \neq q_{m_{\alpha}+1}$ ) implies that there is at least one RFM in the location range  $[\Psi_{\alpha} + 1, m_{\alpha} + 1]$  ( $[\Phi_{\alpha} + 1, m_{\alpha} + 1]$ ). But  $\Psi_{\alpha} \geq m_{\alpha-1} + 1$  ( $\Phi_{\alpha} \geq m_{\alpha-1} + 1$ ). Hence, there is at least one RFM in the location range  $[m_{\alpha-1} + 2, m_{\alpha} + 1]$ , i.e.,  $\text{RFM}_Q[m_{\alpha-1} + 2, m_{\alpha} + 1] \geq 1$ . But by assumption  $\text{RFM}_Q[1, m_{\alpha-1} + 1] = \alpha - 1$ . Hence,  $\text{RFM}_Q[1, m_{\alpha} + 1] \geq \alpha$ . ■

**Lemma 4:** If  $K > 1$ , then for all  $k$ ,  $1 \leq k \leq K - 1$ ,

$$d_{\Psi_k} - d_{\Phi_k} = w - 1.$$

*Proof:* From the algorithm,

$$\max(E_k) - \min(E_k) < w$$

while

$$\max(E_k \cup \{d_{m_k+1}\}) - \min(E_k \cup \{d_{m_k+1}\}) \geq w.$$

Hence, either  $d_{m_k+1} < \min(E_k)$  or  $d_{m_k+1} > \max(E_k)$ .

By Definition 1,  $\{d_{m_k+1} - d_{m_k}\} = 1$ . Since  $d_{m_k} \in E_k$ , either  $d_{m_k+1} = \min(E_k) - 1$  or  $d_{m_k+1} = \max(E_k) + 1$ . Hence,

$$\begin{aligned} \max(E_k \cup \{d_{m_k+1}\}) - \min(E_k \cup \{d_{m_k+1}\}) \\ = \max(E_k) - \min(E_k) + 1. \end{aligned}$$

Thus,  $\max(E_k) - \min(E_k) \geq w - 1$ . But from the algorithm,  $\max(E_k) - \min(E_k) \leq w - 1$ . Hence,

$$\max(E_k) - \min(E_k) = w - 1,$$

i.e.,  $d_{\Psi_k} - d_{\Phi_k} = w - 1$ . ■

**Lemma 5:** If  $K > 1$ , for all  $k$ ,  $1 \leq k \leq K - 1$ , for all  $i$ ,

$$m_{k-1} + 1 \leq i \leq m_k, p_i = d_{\Psi_k} = d_{\Phi_k} - w + 1.$$

For the last subsequence, i.e.,  $k = K$ : if  $\Theta_k = \Phi_k$ , then  $p_i = d_{\Phi_k}$ , else  $p_i = d_{\Psi_k} - w + 1$ .

*Proof:* For all  $i$ ,  $1 \leq i \leq n$ , the value of  $p_i$  is set in step 5 or in step 7 of the algorithm.

If  $1 \leq k \leq K - 1$ , then by Lemma 4,

$$d_{\Psi_k} - d_{\Phi_k} = w - 1.$$

Hence,

$$d_{\Psi_k} = d_{\Phi_k} - w + 1$$

and the same value ( $d_{\Psi_k}$ ) will be assigned to  $p_i$  in step 5 or step 7 of the algorithm.

If  $k = K$ , then it may be the case that  $d_{\Psi_k} - d_{\Phi_k} < w - 1$ . Hence, it may make a difference whether the value of  $p_i$  is assigned in step 5 or in step 7. This is controlled by the value of  $\Theta_k$ .

If  $\Theta_k = \Phi_k$ , then, by the definition of  $\Theta$ ,

$$d_{\Psi_k} < d_{\Phi_{k-1}}.$$

Since  $k - 1 < K$ ,

$$p_{m_{k-1}} = p_{\Phi_{k-1}} = d_{\Phi_{k-1}}.$$

Hence,

$$\min(E_k) < p_{m_{k-1}},$$

and the second clause in step 4 of the algorithm is satisfied. Thus,  $p_i$  ( $m_{k-1} + 1 \leq i \leq m_k$ ) is assigned a value in step 5 of the algorithm. So

$$p_i = d_{\Psi_k}.$$

If  $\Theta_k = \Psi_k$ , then, by the definition of  $\Theta$ ,

$$d_{\Psi_k} \geq d_{\Phi_{k-1}}.$$

Since  $k - 1 < K$ ,

$$p_{m_{k-1}} = p_{\Phi_{k-1}} = d_{\Phi_{k-1}}.$$

Hence,

$$\min(E_k) \geq p_{m_{k-1}}.$$

and the second clause in step 4 of the algorithm is *not* satisfied. Since  $k = K$ ,  $m_k = n$ , and the first clause in step 4 of the algorithm is also *not* satisfied. Thus,  $p_i$  ( $m_{k-1} + 1 \leq i \leq m_k$ ) is assigned a value of step 7 of the algorithm. So

$$p_i = d_{\Psi_k} - w + 1.$$

*Lemma 6:* If  $K > 1$ , then for all  $k$ ,  $1 \leq k \leq K$ .

$$MT_Q[1, O_k] \geq MT_P[1, O_k] + |q_{O_k} - p_{O_k}|.$$

*Proof:* By induction on  $k$ .

*Basis:*  $k = 1$ . It is shown that  $MT_Q[1, O_1] \geq MT_P[1, O_1] + |q_{O_1} - p_{O_1}|$ .

By the definition of  $O$ ,  $O_1 = 1$ . Hence,

$$MT_Q[1, O_1] = MT_Q[1, 1] = |q_1 - q_0|$$

and

$$\begin{aligned} MT_P[1, O_1] + |q_{O_1} - p_{O_1}| &= MT_P[1, 1] + |q_1 - p_1| \\ &= |p_1 - p_0| + |q_1 - p_1|. \end{aligned}$$

By Definitions 2 and 5, for all  $i$ ,  $1 \leq i \leq n$ ,

$$1 \leq q_i \leq d_i < q_i + w$$

and

$$1 \leq p_i \leq d_i < p_i + w.$$

By Definition 1,  $d_1 = 1$ . Hence,  $q_1 = p_1 = 1$ . From the algorithm,  $p_0 = 1$ . Hence,

$$MT_P[1, O_1] + |q_{O_1} - p_{O_1}| = |p_1 - p_0| + |q_1 - p_1| = 0.$$

Since  $|q_1 - q_0| \geq 0$ ,

$$MT_Q[1, O_1] \geq 0.$$

Thus,

$$MT_Q[1, O_1] \geq MT_P[1, O_1] + |q_{O_1} - p_{O_1}|.$$

*Induction Step:* Assuming that this lemma holds for  $k = \alpha - 1$ , where  $2 \leq \alpha \leq K$ , it is now proven that it holds for  $k = \alpha$ . In other words, assuming

$$MT_Q[1, O_{\alpha-1}] \geq MT_P[1, O_{\alpha-1}] + |q_{O_{\alpha-1}} - p_{O_{\alpha-1}}|,$$

it is proven that

$$MT_Q[1, O_\alpha] \geq MT_P[1, O_\alpha] + |q_{O_\alpha} - p_{O_\alpha}|.$$

From Definition 8,

$$MT_Q[1, O_\alpha] = MT_Q[1, O_{\alpha-1}] + MT_Q[O_{\alpha-1} + 1, O_\alpha]$$

and

$$MT_P[1, O_\alpha] = MT_P[1, O_{\alpha-1}] + MT_P[O_{\alpha-1} + 1, O_\alpha].$$

Using the induction hypothesis,

$$\begin{aligned} MT_Q[1, O_\alpha] &\geq MT_P[1, O_{\alpha-1}] + |q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| \\ &\quad + MT_Q[O_{\alpha-1} + 1, O_\alpha]. \end{aligned}$$

$MT_Q[O_{\alpha-1} + 1, O_\alpha]$  is the number of STACK1 frames transferred to/from memory in location range  $[O_{\alpha-1} + 1, O_\alpha]$ . A change by one in the RFP indicates that one STACK1 frame is transferred to or from memory. Hence, the memory traffic in location range  $[O_{\alpha-1} + 1, O_\alpha]$  is at least the difference between the RFP at the beginning of the range and the RFP at the end of the range, i.e.,

$$MT_Q[O_{\alpha-1} + 1, O_\alpha] \geq |q_{O_\alpha} - q_{O_{\alpha-1}}|.$$

Hence,

$$MT_Q[1, O_\alpha] \geq MT_P[1, O_{\alpha-1}] + |q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| + |q_{O_\alpha} - q_{O_{\alpha-1}}|.$$

From Definition 8 and the algorithm,

$$\begin{aligned} MT_P[O_{\alpha-1} + 1, O_\alpha] &= \sum_{\beta=O_{\alpha-1}+1}^{O_\alpha} |p_\beta - p_{\beta-1}| \\ &= \sum_{\beta=O_{\alpha-1}+1}^{m_{\alpha-1}} |p_\beta - p_{\beta-1}| + |p_{m_{\alpha-1}+1} - p_{m_{\alpha-1}}| \\ &\quad + \sum_{\beta=m_{\alpha-1}+2}^{O_\alpha} |p_\beta - p_{\beta-1}| = |p_{m_{\alpha-1}+1} - p_{m_{\alpha-1}}|. \end{aligned}$$

Since  $\alpha - 1 < K$ , by Lemma 5,  $p_{m_{\alpha-1}} = d_{\Phi_{\alpha-1}}$ . From the algorithm,  $p_{m_{\alpha-1}+1} = p_{m_\alpha}$ . Hence,

$$MT_P[O_{\alpha-1} + 1, O_\alpha] = |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

Thus,

$$MT_P[1, O_\alpha] = MT_P[1, O_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

In the rest of the proof, the following four cases will be handled separately:

Case A:  $O_\alpha = \Phi_\alpha$  and  $O_{\alpha-1} = \Phi_{\alpha-1}$

Case B:  $O_\alpha = \Phi_\alpha$  and  $O_{\alpha-1} = \Psi_{\alpha-1}$

Case C:  $O_\alpha = \Psi_\alpha$  and  $O_{\alpha-1} = \Phi_{\alpha-1}$

Case D:  $O_\alpha = \Psi_\alpha$  and  $O_{\alpha-1} = \Psi_{\alpha-1}$ .

Case A:  $O_\alpha = \Phi_\alpha$  and  $O_{\alpha-1} = \Phi_{\alpha-1}$ :

$$\begin{aligned} &|q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| + |q_{O_\alpha} - q_{O_{\alpha-1}}| \\ &= |q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}}| + |q_{\Phi_\alpha} - q_{\Phi_{\alpha-1}}| \\ &= |p_{\Phi_{\alpha-1}} - q_{\Phi_{\alpha-1}}| + |q_{\Phi_{\alpha-1}} - q_{\Phi_\alpha}| \geq p_{\Phi_{\alpha-1}} - q_{\Phi_{\alpha-1}} \\ &\quad + q_{\Phi_{\alpha-1}} - q_{\Phi_\alpha} = p_{\Phi_{\alpha-1}} - q_{\Phi_\alpha} \\ &= (p_{\Phi_{\alpha-1}} - p_{\Phi_\alpha}) + (p_{\Phi_\alpha} - q_{\Phi_\alpha}). \end{aligned}$$

By Lemma 5, since  $O_\alpha = \Phi_\alpha$  and  $\alpha - 1 < K$ ,  $p_{\Phi_\alpha} = d_{\Phi_\alpha}$  and  $p_{\Phi_{\alpha-1}} = d_{\Phi_{\alpha-1}}$ . Hence,

$$\begin{aligned} &|q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| + |q_{O_\alpha} - q_{O_{\alpha-1}}| \\ &\geq (d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}) + (d_{\Phi_\alpha} - q_{\Phi_\alpha}). \end{aligned}$$

Since  $Q$  is a valid RFPS for  $D$ ,

$$q_{\Phi_\alpha} \leq d_{\Phi_\alpha} < q_{\Phi_\alpha} + w.$$

Hence,  $(d_{\Phi_\alpha} - q_{\Phi_\alpha}) \geq 0$ . Thus,

$$|q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| + |q_{O_\alpha} - q_{O_{\alpha-1}}| \geq d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}.$$

From the definition of  $O$ , since  $O_\alpha = \Phi_\alpha$ ,  $d_{\Phi_{\alpha-1}} > d_{\Phi_\alpha}$ . Hence,

$$d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha} = |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|.$$

Thus,

$$|q_{O_{\alpha-1}} - p_{O_{\alpha-1}}| + |q_{O_\alpha} - q_{O_{\alpha-1}}| \geq |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|.$$

Therefore,

$$MT_Q[1, O_\alpha] \geq MT_P[1, O_{\alpha-1}] + |d_{\Phi_{\alpha-1}} - d_{\Phi_\alpha}|.$$

By Lemma 5, since  $O_\alpha = \Phi_\alpha$ ,  $p_{m_\alpha} = d_{\Phi_\alpha}$ . Hence,

$$MT_P[1, O_\alpha] = MT_P[1, O_{\alpha-1}] + |d_{\Phi_\alpha} - d_{\Phi_{\alpha-1}}|.$$

Thus,

$$MT_Q[1, \Theta_\alpha] \geq MT_P[1, \Theta_\alpha].$$

Case B:  $\Theta_\alpha = \Phi_\alpha$  and  $\Theta_{\alpha-1} = \Psi_{\alpha-1}$ :

$$\begin{aligned} |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \\ = |q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}}| + |q_{\Phi_\alpha} - q_{\Psi_{\alpha-1}}| \\ = |p_{\Psi_{\alpha-1}} - q_{\Psi_{\alpha-1}}| + |q_{\Psi_{\alpha-1}} - q_{\Phi_\alpha}| \geq p_{\Psi_{\alpha-1}} \\ - q_{\Psi_{\alpha-1}} + q_{\Psi_{\alpha-1}} - q_{\Phi_\alpha} = p_{\Psi_{\alpha-1}} - q_{\Phi_\alpha}. \end{aligned}$$

From the algorithm,  $p_{\Psi_{\alpha-1}} = p_{\Phi_{\alpha-1}}$ . Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq p_{\Phi_{\alpha-1}} - q_{\Phi_\alpha}.$$

The rest of the proof for this case is identical to the proof of Case A.

Case C:  $\Theta_\alpha = \Psi_\alpha$  and  $\Theta_{\alpha-1} = \Phi_{\alpha-1}$ :

$$\begin{aligned} |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \\ = |q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}}| + |q_{\Psi_\alpha} - q_{\Phi_{\alpha-1}}| \\ \geq q_{\Phi_{\alpha-1}} - p_{\Phi_{\alpha-1}} + q_{\Psi_\alpha} - q_{\Phi_{\alpha-1}} = q_{\Psi_\alpha} - p_{\Phi_{\alpha-1}} \\ = (q_{\Psi_\alpha} - p_{\Phi_\alpha}) + (p_{\Phi_\alpha} - p_{\Phi_{\alpha-1}}). \end{aligned}$$

By Lemma 5, since  $\Theta_\alpha = \Psi_\alpha$  and  $\alpha - 1 < K$ ,  $p_{\Phi_\alpha} = d_{\Psi_\alpha} - w + 1$  and  $p_{\Phi_{\alpha-1}} = d_{\Phi_{\alpha-1}}$ . Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq (q_{\Psi_\alpha} - d_{\Psi_\alpha} + w - 1) + (d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}).$$

Since  $Q$  is a valid RFPS for  $D$ ,

$$q_{\Psi_\alpha} \leq d_{\Psi_\alpha} < q_{\Psi_\alpha} + w.$$

Hence,  $q_{\Psi_\alpha} - d_{\Psi_\alpha} + w > 0$ , so

$$q_{\Psi_\alpha} - d_{\Psi_\alpha} + w - 1 \geq 0.$$

Thus,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}.$$

From the definition of  $\Theta$ , since  $\Theta_\alpha = \Psi_\alpha$ ,  $d_{\Phi_\alpha} > d_{\Phi_{\alpha-1}}$ . From the algorithm,

$$\max(E_{\alpha-1}) - \min(E_{\alpha-1}) < w$$

while

$$\max(E_{\alpha-1} \cup \{d_{m_{\alpha-1}+1}\}) - \min(E_{\alpha-1} \cup \{d_{m_{\alpha-1}+1}\}) \geq w.$$

Hence, either  $d_{m_{\alpha-1}+1} < d_{\Phi_{\alpha-1}}$  or  $d_{m_{\alpha-1}+1} > d_{\Psi_{\alpha-1}}$ . In this case, since  $d_{\Phi_\alpha} > d_{\Phi_{\alpha-1}}$  and  $d_{m_{\alpha-1}+1} \geq d_{\Phi_\alpha}$ , it must be true that  $d_{m_{\alpha-1}+1} > d_{\Psi_{\alpha-1}}$ . By the definition of  $\Psi$ ,  $d_{\Psi_\alpha} \geq d_{m_{\alpha-1}+1}$ . Hence,

$$d_{\Psi_\alpha} > d_{\Psi_{\alpha-1}}.$$

By Lemma 4, since  $\alpha - 1 < K$ ,  $d_{\Psi_{\alpha-1}} = d_{\Phi_{\alpha-1}} + w - 1$ . Hence,  $d_{\Psi_\alpha} > d_{\Phi_{\alpha-1}} + w - 1$ , so

$$d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}} > 0.$$

Thus,

$$d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}} = |d_{\Psi_\alpha} - w + 1 - d_{\Phi_{\alpha-1}}|.$$

By Lemma 5, since  $\Theta_\alpha = \Psi_\alpha$ ,  $p_{m_\alpha} = d_{\Psi_\alpha} - w + 1$ . Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

Therefore,

$$MT_Q[1, \Theta_\alpha] \geq MT_P[1, \Theta_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|.$$

It has been shown above that  $MT_P[1, \Theta_\alpha] = MT_P[1, \Theta_{\alpha-1}] + |p_{m_\alpha} - d_{\Phi_{\alpha-1}}|$ . Hence,

$$MT_Q[1, \Theta_\alpha] \geq MT_P[1, \Theta_\alpha].$$

Case D:  $\Theta_\alpha = \Psi_\alpha$  and  $\Theta_{\alpha-1} = \Psi_{\alpha-1}$ :

$$\begin{aligned} |q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \\ = |q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}}| + |q_{\Psi_\alpha} - q_{\Psi_{\alpha-1}}| \\ \geq q_{\Psi_{\alpha-1}} - p_{\Psi_{\alpha-1}} + q_{\Psi_\alpha} - q_{\Psi_{\alpha-1}} = q_{\Psi_\alpha} - p_{\Psi_{\alpha-1}}. \end{aligned}$$

From the algorithm,  $p_{\Psi_{\alpha-1}} = p_{\Phi_{\alpha-1}}$ . Hence,

$$|q_{\Theta_{\alpha-1}} - p_{\Theta_{\alpha-1}}| + |q_{\Theta_\alpha} - q_{\Theta_{\alpha-1}}| \geq q_{\Psi_\alpha} - p_{\Phi_{\alpha-1}}.$$

The rest of the proof for this case is identical to the proof of Case C. ■

#### ACKNOWLEDGMENT

We would like to thank P. Denning, D. Ferrari, M. Katevenis, J. Ousterhout, R. Sherburne, and A. Smith for their useful suggestions on improving this paper and D. Patterson for his help with the development of some of the initial RISC analysis tools.

#### REFERENCES

- [1] L. A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78-101, 1966.
- [2] R. Campbell, "A C compiler for RISC," M.S. rep., Univ. California, Berkeley, Dec. 1980.
- [3] P. J. Denning, private communication, May 1982.
- [4] *VAX11 Architecture Handbook*, Digital Equipment Corp., 1979.
- [5] D. Halbert and P. Kessler, "Windows of overlapping register frames," in *CS292R Final Project Reports* (unpublished), Univ. California, Berkeley, June 1980, pp. 82-100.
- [6] S. C. Johnson, "A portable compiler: Theory and practice," in *Proc. 5th ACM Symp. Principles of Programming Languages*, Jan. 1978, pp. 97-104.
- [7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
- [8] D. A. Patterson and C. H. Séquin, "RISC I: A reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. Comput. Architecture*, Minneapolis, MN, May 1981, pp. 443-457.
- [9] —, "A VLSI RISC," *Computer*, vol. 15, pp. 8-21, Sept. 1982.
- [10] A. J. Smith, "Cache memories," *Comput. Surveys*, vol. 14, pp. 473-530, Sept. 1982.
- [11] Y. Tamir, "Simulation and performance evaluation of the RISC architecture," Electron. Res. Lab., Univ. California, Berkeley, Memo. UCB/ERL, M81/17, Mar. 1981.



Yuval Tamir (S'78) received the B.S.E.E. degree ("with highest distinction") from the University of Iowa, Iowa City, in 1979 and the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, in 1981.

Since 1979 he has been a Research Assistant in the Electronics Research Laboratory at U.C. Berkeley where he is currently working on his Ph.D. dissertation. His research interests are fault-tolerant computing, computer architecture, and distributed systems.

Mr. Tamir is a student member of the IEEE Computer Society and the Association for Computing Machinery.





**Carlo H. Séquin** (M'71-SM'80-F'82) received the Ph.D. degree in experimental physics from the University of Basel, Basel, Switzerland, in 1969.

In 1969-1970 he performed postdoctoral work at the Institute of Applied Physics, University of Basel, which concerned interface physics of MOS transistors and problems of applied electronics in the field of cybernetic models. From 1970 to 1976 he worked at Bell Laboratories, Murray Hill, NJ, in the MOS Integrated Circuit Laboratory on the design and investigation of charge-coupled devices for imaging and signal processing applications. He spent 1976-1977 on leave of absence with the University of California, Berkeley, where he lectured on integrated circuits,

logic design, and microprocessors. In 1977 he joined the faculty in the Department of Electrical Engineering and Computer Sciences, where he is Professor of Computer Science. Since 1980 he has headed the CS Division as Associate Chairman for Computer Sciences. His research interests lie in the field of computer architecture and design tools for very large scale integrated systems. In particular, his research concerns multimicroprocessor computer networks, the mutual influence of advanced computer architectures and modern VLSI technology, and the implementation of special functions in silicon. Since 1977 he has been teaching courses in structured MOS-LSI design. He is an author of the first book on charge-transfer devices, and has written many papers in that field.

Dr. Séquin is a member of the Association for Computing Machinery and the Swiss Physical Society.

## **LU Decomposition on a Multiprocessing System with Communications Delay**

**Wang Ho Yu**

**Ph.D.**

**Electrical Engineering  
and Computer Sciences**

**Sponsor:  
Defense Advanced Research Project Agency**

---

**D.G. Messerschmitt  
Chairman of Committee**

### **ABSTRACT**

A large amount of computer time is used for the solution of systems of linear equations in the course of the circuit simulation during the design of integrated circuits. This expenditure limits the size of circuits which can be practically simulated, and results in poor response time in an interactive environment. In order to increase the size of circuits which can be simulated, and increase the response time, one option pursued here is to apply concurrent computation to the linear equation solution aspect of circuit simulation. This concurrent computation will exploit inherent parallelism in the linear equation solution to reduce the time required for that solution. We focus on one particular method for solution of the linear equations: LU decomposition.

While LU decomposition has a great deal of inherent parallelism, the wide range of sparse matrix structures requires that this parallelism be detected automatically. It has been determined that the overall speedup is sensitive to the delays between cooperating computational elements, and the manner in which the concurrent computations are mapped onto computational elements is therefore of importance. The approach used is as follows : Given a sparse matrix

with a particular structure, a code generator produces a program representing the LU decomposition for that matrix. Another program detects the precedence constraints among the sequential instructions in the code and models the solution process as a directed graph. Based on this graph, scheduling techniques are employed to assign segments of code to computational elements for concurrent execution.

Most of this thesis concentrates on the last problem, finding scheduling algorithms which reduce the sensitivity of the solution time to the communication delay among computational elements. This is based on the following observation. With zero delay, the common Hu's level scheduling algorithm gives good speedup performance. However when the communication delay is large compared to the execution time of an instruction in the code, considerable degradation on the speedup performance is observed for Hu's algorithm.

Polynomial-time optimal scheduling algorithms appear to be intractable. Hence heuristic algorithms with feasible running time that give suboptimal schedules have to be constructed. This is approached in two different ways. Heuristic *local minimization* scheduling algorithms using two matching algorithms from combinatorial optimization are studied and promising results are obtained. These two matching algorithms, min\_max matching and weighted matching, give optimal code-to-processor assignment at each time step. The second approach is heuristic *global minimization* using a clustering technique. The critical (longest) path in the directed graph has a close correlation with the completion time. The idea is to shorten the critical path by clustering nodes together in order to reduce the communication delay.

Experimental results are given for both approaches based on the solution of a set of sparse linear equations based on actual circuit simulations.

## ACKNOWLEDGEMENT

I would like to express my sincere appreciation to my research advisor, Professor David G. Messerschmitt, for his valuable inspiration, continuing guidance and encouragement throughout the years of my research at UC Berkeley. Without his support, none of this would have been possible. I wish to thank Richard Fujimoto, Hui-Hung Lu and Po Tong for their useful discussions and suggestions. The support from the Defense Advanced Research Project Agency is acknowledged.

I am also grateful to my family for their constant encouragement and spiritual support. Last, but not the least, I wish to thank my wife, Yuk-Pai, for her love, care and understanding through most of the course of this research.

## Table of Contents

<b>Chapter 1 : Introduction .....</b>	<b>1</b>
1.1 Motivation and the Goal of this Research .....	1
1.2 Brief Review of Standard Circuit Simulator .....	2
1.3 Impact of Parallel Processing on Circuit Simulation .....	4
1.3.1 Decomposition .....	4
1.3.2 Advantages of Decomposition .....	5
1.4 Simulation Tool .....	6
1.4.1 The Simulator, SIMON .....	6
1.4.2 Important Features of the Simulator .....	7
1.5 Problem Statement .....	7
1.6 Outline of the Dissertation .....	8
<b>Chapter 2 : Partition of LU Decomposition for Concurrent Processing .....</b>	<b>13</b>
2.1 Introduction .....	13
2.2 Doolittle Algorithm for LU Decomposition .....	14
2.2.1 Forward and Backward Substitutions .....	14
2.2.2 Doolittle Algorithm .....	15
2.3 Code Generation .....	17
2.3.1 The Code Generator .....	18
2.3.2 Simple Example of the Code Generated .....	19
2.4 Graph Model for LU Decomposition .....	20
2.4.1 Detection of Precedence Constraints .....	20
2.4.2 Implementation of Detection of Precedence Constraints .....	21
2.4.3 The LU Graph Model .....	22
2.4.4 Data Structure for LU Task Graph .....	26
2.5 Processor Scheduling Techniques .....	27
2.5.1 Performance Criteria of a Schedule and Efficiency of an Algorithm .....	28
2.6 Extension to Other Algorithms .....	30
2.6.1 The Inner Product .....	30
2.6.2 Linear Convolution .....	31
2.6.3 Fast Fourier Transform .....	32
2.7 Conclusion .....	33
<b>Chapter 3 : Scheduling Algorithms Without Communication Delay                 Consideration .....</b>	<b>41</b>
3.1 Introduction .....	41
3.2 List Schedules .....	42
3.3 Hu's Level Scheduling Algorithm .....	43

3.3.1	Minimum Time Required to Finish the Task Graph .....	46
3.3.2	Maximum Number of Processors Required .....	46
3.3.3	Minimum Number of Processors Required .....	46
3.3.4	Maximum Achievable Speedup Ratio .....	47
3.4	Impact of Communication Delay on Speedup Performance .....	47
3.4.1	Components of Communication Delay .....	48
3.4.2	Determination of Completion Time of a LU Task Graph .....	50
3.5	Results and Discussion .....	51
3.6	Conclusion .....	53
<b>Chapter 4 : Heuristic Local Scheduling Techniques .....</b>		<b>63</b>
4.1	Introduction .....	63
4.2	Complexity and Difficulty of the Problem .....	64
4.3	Definitions and Terminologies .....	65
4.4	Heuristic Algorithm D .....	68
4.5	Application of Min_Max and Weighted Matchings to Heuristic Algorithms .....	72
4.5.1	Simple Illustrative Example .....	73
4.6	Heuristic Algorithm E .....	75
4.7	Heuristic Algorithm F .....	77
4.8	Heuristic Algorithm EF .....	79
4.9	Complexity of these Heuristic Algorithms .....	80
4.10	Simulation Results and Discussion .....	81
4.11	Conclusion .....	83
<b>Chapter 5 : Heuristic Global Clustering Technique .....</b>		<b>114</b>
5.1	Introduction .....	114
5.2	Finding the Critical Path in a Directed Graph .....	114
5.2.1	Dynamic Programming Approach .....	116
5.2.2	The Cascade Algorithm .....	116
5.3	Modification of the Cascade Algorithm to find Critical Paths .....	119
5.3.1	Data Structure for Matrices D and R .....	120
5.4	Application of Modified Cascade Algorithm to LU Task Graph .....	121
5.5	Clustering Technique .....	123
5.5.1	A Modified Heuristic Approach .....	123
5.6	High Level Description of the Heuristic Clustering Technique .....	126
5.6.1	Data Structure for Managing the Clusters .....	127
5.7	Complexity of the Heuristic Clustering Algorithm .....	129
5.8	Simulation Results .....	130
5.8.1	Reduction of the Length of the Critical Path .....	130
5.8.2	Reduction on the Number of Nodes .....	131
5.8.3	Comparison of Global and Local Heuristic Techniques .....	132

5.9	Conclusion .....	134
<b>Chapter 6 : Conclusion</b>	.....	150
6.1	Overall Review of the Problem .....	150
6.2	Significance of this Research .....	151
6.3	Future Related Research .....	151
<b>References</b>	.....	154



## INTRODUCTION

## 1.1. Motivation and the Goal of this Research

In the domain of circuit simulation, practically all standard circuit simulators such as SPICE[1] and ASTAP[2] contain a routine for solving a system of linear equations. For small size circuits, most of the cpu time is spent in loading the circuit matrix and the model evaluations. This processing time grows linearly with the size of the circuits. For large circuits, it has been observed that a large portion of the cpu simulation time is spent in solving the system of linear equations. It has been estimated that this linear equations solution time grows as  $O(n^\beta)$  where  $n$  is the size of the circuit measured in the number of circuit elements and  $\beta$  is between 1.1 and 1.5. Hence for large circuits, this will become the dominant cost of the analysis.

In order to reduce the simulation time of large circuits, decomposition of the original problem into smaller modules for parallel computing should be considered. These modules will be assigned to different processors for concurrent execution. Due to the inherent parallelism in the solution algorithm of the system of linear equations, this concurrency technique will speed up the solution process.

In the most general case, the precedence constraints among these modules are arbitrary. In this case, the assignment of these modules to processors to obtain a schedule which has the shortest possible finishing time is very complicated. Another consideration is that there exists a nonzero interprocessor communication overhead in any realistic distributed computing environment. This overhead will tend to degrade the speedup performance of the multiprocessing

system. Hence this overhead delay makes the already very complicated problem even more difficult. In this dissertation, the goal is to construct some heuristic scheduling algorithms taking into account the communication delay between processors, so that the finishing time of a schedule is as small as possible.

## 1.2. Brief Review of Standard Circuit Simulator

In the design of integrated circuits, circuit simulation programs such as SPICE [1] and ASTAP [2] are shown to give very accurate analysis of electrical characteristics of a circuit. With these computer-aided design tools, circuit designers can verify and modify the circuit to get a better design with great flexibility. This approach virtually replaces the traditional breadboard and testing approach as a means of verifying the electrical performance of the final circuit.

In most of the cases, the simulation of an electrical circuit requires three basic types of analyses: the dc analysis, the small signal ac analysis and the transient analysis in the time-domain. In the most general case of transient analysis, the dynamical behaviour of the circuit is described by a system of differential equations

$$f(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{u}(t)) = 0; \quad \mathbf{x}(0) = \mathbf{x}_0 \quad (1.1)$$

where  $\mathbf{x}(t) \in \mathbb{R}^n$  is the vector of unknown circuit variables,  $\mathbf{u}(t) \in \mathbb{R}^p$  is the vector of input circuit variables and their time derivatives,  $\mathbf{x}_0 \in \mathbb{R}^n$  is the given initial value of  $\mathbf{x}$ , and  $f$  is a vector-valued continuous function. The simulation is carried out through a sequence of discrete timesteps  $t_i; i = 1, 2, \dots, N$  chosen by the simulator with  $t_0 = 0$  and  $t_N = T$  where  $T$  is the simulation time specified by the user.

The three basic procedures used to solve this system over the time interval  $[0, T]$  are : an implicit numerical integration scheme, the Newton-Raphson algo-

rithm and the solution of the resulting system of linear equations. The differential equations describing the reactive elements are replaced by their corresponding *discrete circuit models* associated with an implicit integration algorithm such as Backward Euler

$$\dot{x}(t_i) \approx \frac{x(t_i) - x(t_{i-1})}{t_i - t_{i-1}} \quad (1.2)$$

or Trapezoidal rule

$$\dot{x}(t_i) \approx 2 \frac{x(t_i) - x(t_{i-1})}{t_i - t_{i-1}} - \dot{x}(t_{i-1}) \quad (1.3)$$

At this stage, we have a resistive network consisting of linear and/or nonlinear elements. The nonlinear resistive elements are then substituted by their *companion* models using Newton-Raphson algorithm.

$$\frac{\partial g}{\partial x}(x_i^{k-1})[x_i^k - x_i^{k-1}] = -g(x_i^{k-1}) \quad (1.4)$$

where  $g(\ )$  is the branch relation describing the nonlinear element,  $\frac{\partial g}{\partial x}(x_i^{k-1})$  is the partial derivative of  $g(\ )$  evaluated at  $x_i^{k-1}$ , and  $k$  is the Newton-Raphson iteration count. The resulting network contains only linear resistive elements. Modified nodal analysis (or Sparse Tableau) is used to assemble the linear circuit equations and the solution of these equations is sought by LU decomposition followed by forward and backward substitutions (Gaussian Elimination). The coefficient matrix obtained is usually very sparse and for efficient solution of the linear circuit equations, sparse matrix techniques are employed [3]. These procedures are repeated for each time step until the whole time interval of simulation is completed. A more detail analysis and description of the above algorithms are found in [4]. The time-domain transient analysis can be summarized in the flow chart shown in Figure 1.1

It has been observed in [5] that the bulk of the storage and computation of the simulation lie in loading the modified nodal analysis matrix and solving the

linearized circuit equations. In particular, for very large circuits, the computation time spent in the solution of the system of linear circuit equations grows exponentially as  $n^k$  where  $k$  is between 1.1 and 1.5 and  $n$  is the size of the circuit measured in terms of circuit components. Hence, for cost-effective use of the simulator, the simulation is usually limited to circuits of a few hundred devices.

### 1.3. Impact of Parallel Processing on Circuit Simulation

As the results of improvements in fabrication and processing technologies, we are moving into the era of very large scale integrated (VLSI) circuits. The natural consequence of this is the increase in demand for simulating these VLSI circuits. It is not unusual to find large mainframe computers dedicated solely to circuit simulation in some major integrated circuit design houses.

#### 1.3.1. Decomposition

As already pointed out in the last paragraph of section 1.2, it is not cost-effective to simulate the whole circuit on a single computer. Hence, new simulation techniques are necessary in order to cope with the problems suffered in the standard circuit simulators. A survey of the third generation simulator algorithms is found in [6]. These algorithms are based on the concept of decomposition of large-scale systems. Within this context, decomposition refers to the partitioning of the problem of solving a system of equations into many subproblems. Each subproblem consists of a subset of the original equations and the corresponding variables. The solution of the original problem is obtained by considering the interactions between these subproblems. Some of the well known decomposition algorithms are Block LU Factorization, the Tearing Algorithm, the Multilevel Newton-Raphson Algorithm and Relaxation Algorithm ( see[7] and[8] ). In this dissertation, we have another decomposition technique which is different from the traditional decomposition algorithms mentioned above. We call it "ele-

mental" decomposition[9] , which exploits the parallelism among the elemental operations representing a particular algorithm. Here the decomposition is done on the individual operations.

### 1.3.2. Advantages of Decomposition

With the advances in the VLSI technology and the price-performance improvements in hardware, we can expect to be able to perform computationally intensive computation on a multiprocessor computing system in the near future. The realization of the potential offered by these special concurrent computer architectures lies on the development of new parallel computational algorithms such as the ones mentioned above. No matter which decomposition technique is employed, the most important advantage of decomposition is the capability to use concurrency to increase the size of the circuit to be simulated. We envision a multiprocessor system consisting of a set of processors and an interconnection. Each processor has its own memory and the processors can only communicate through a central interconnection network. In other words, all the communication functions such as routing, message forwarding, buffer management are done by the nodes within the interconnection network. A conceptual model of a multiprocessor system is shown in Figure 1.2. The advantages of decomposition in the context of parallel processing are as follows :

#### *High Speed Capability*

The subproblems or modules of a decomposed system can run on processors which work cooperatively to gain high speed through concurrent execution and hence the response time is reduced. An additional gain in speed would result from the smaller memory addressed within each processor.[10].

#### *Memory Size*

As pointed out in section 1.2, the storage required for a simulation of VLSI circuits is very large and hence it is not practical to simulate the whole circuit on a

single computer. By partitioning the original circuit into modules, these modules will be treated on separate processors and this permits much larger circuit to be simulated.

### *Latency*

One of the advantages of a distributed computing system is the ability to share resources available at various computing components. By exploiting the latency of the circuit i.e. avoiding the expense of simulating parts of the circuit which are not active at a given point in the simulation time, we can redirect some of the computing resources available for other useful purposes.

This divide-and-conquer approach is not only appealing for circuit simulation, but it also has applications in areas such as real time command and control, data base management and real time signal processing.

## **1.4. Simulation Tool**

In order to evaluate the speedup performance as well as the efficiency of a multiprocessor system when a particular parallel algorithm is executing on it, either a hardwired multiprocessor or a simulator is necessary. Since we don't have a multiprocessor, a discrete-time, event-driven simulation program has been written ([11], and [12]) This simulator is named SIMON (Simulator of Multiprocessor Networks) which simulates the parallel execution of a set of user's programs as if each program is run on a separate processor.

### **1.4.1. The Simulator, SIMON**

The simulator consists of three components as shown in Figure 1.3 : the application programs or equivalently tasks (processes), the simulator base, and the switch model. The application programs are assumed to be written in C for easy interfacing with SIMON. The simulator base time-multiplexes the execution of the processes on a host computer which is a VAX 11/780. The base also keeps

track of the time for each task to ensure that the interactions among these tasks are simulated in the correct time sequence. The switch model models the interconnection network and it facilitates the easy comparison of different switching structures by having this switch model as a separate module.

#### **1.4.2. Important Features of the Simulator**

The most important features of the simulator for the entire simulation can be summarized below:

- (1) provides statistics such as blocked time, running time on each processor.
- (2) gives average traffic measured in terms of number of bytes that has been exchanged between each pair of processors.
- (3) allows the user to model the speed of the processor by specifying the execution times of assembler instructions (comparable to those available in the VAX).
- (4) permits the user to specify a constant time message delay between any pair of communicating processors.

These will provide crucial information about the speedup performance of a parallel algorithm on a switching structure, so that the user can tailor the topology of the interconnection network and is able to design an appropriate message routing algorithm for specific applications such as circuit simulation and signal processing.

#### **1.5. Problem Statement**

With the advantages of concurrent processing applied to the parallel algorithms outlined above, we attempt to implement the LU decomposition on a multiprocessing system with consideration of communication delay in the switching network. The main objective is to achieve a reduction in execution time when we



use a multiprocessor approach as compared to a single processor. This time reduction is measured in terms of speedup performance which is defined as the ratio of the completion time of a given task using multiple processors to the completion time using single processor. In this dissertation, the communication delay is defined as the time elapsed when a processor sends a message and when the message arrives at the destination processor. A further simplification is that a constant delay is assumed for any communicating processors in the multiprocessor system.

Previous work[9] showed a promising speedup performance when the communication delay is ignored and a scheduling algorithm called Hu's level scheduling algorithm is employed. However when the communication delay is large, it will be shown that there is a considerable degradation on the speedup performance. Hence the problem of assigning tasks to processors taking into account the delay to minimize the completion time is of main concern in this research. This belongs to the classical problem of scheduling theory in resource management. In the context of this problem, polynomial-time optimal scheduling algorithms appear to be intractable. Heuristic scheduling algorithms with feasible running time that give suboptimal schedules have to be constructed. The performance of these heuristics in terms of the speedup ratio obtained will be compared to Hu's level scheduling algorithm in the presence of communication delay in the interconnection network.

#### 1.6. Outline of the Dissertation

In the following chapters, the approach to this research and the heuristic scheduling algorithms will be discussed in detail, followed by a concluding chapter of remarks and future research. More precisely, chapter two is devoted to LU decomposition and our approach to implementing the decomposition on a multiprocessor system. Chapter three will give a detailed description of Hu's

level scheduling algorithm applied to the LU decomposition task graph. Then the simulation studies of Hu's level scheduling technique are discussed in the presence of communication delay in the switching network. In chapters four and five, heuristic algorithms taking into account the communication delay will be presented. With respect to this topic of scheduling algorithms, the heuristic local minimization algorithms using the min\_max matching and weighted matching algorithms are discussed in chapter four, and the heuristic global minimization algorithm is detailed in chapter five. In both chapters, simulation results are presented to show the performance improvement of these heuristics over Hu's level scheduling algorithm in the presence of communication delay.

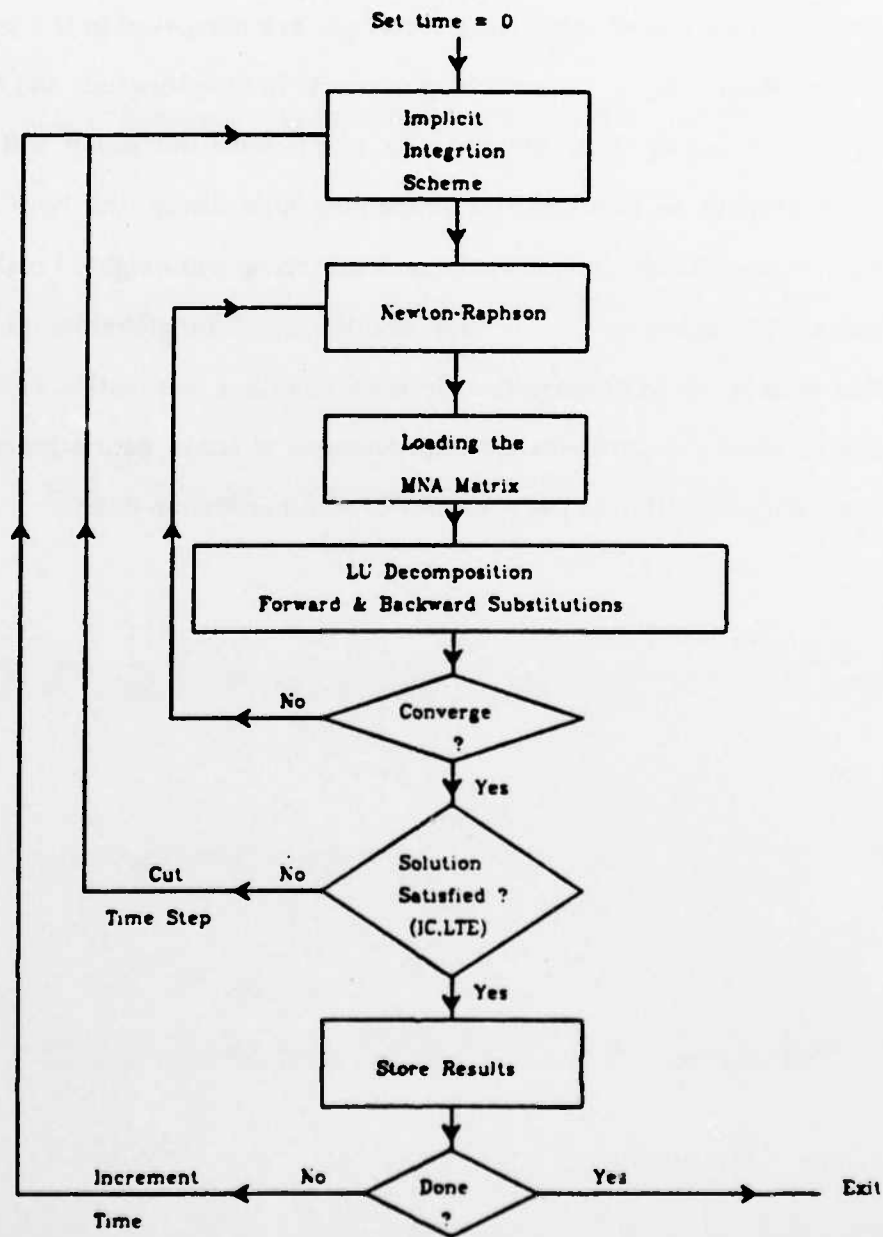


Figure 1.1 Flow Chart for Transient Analysis

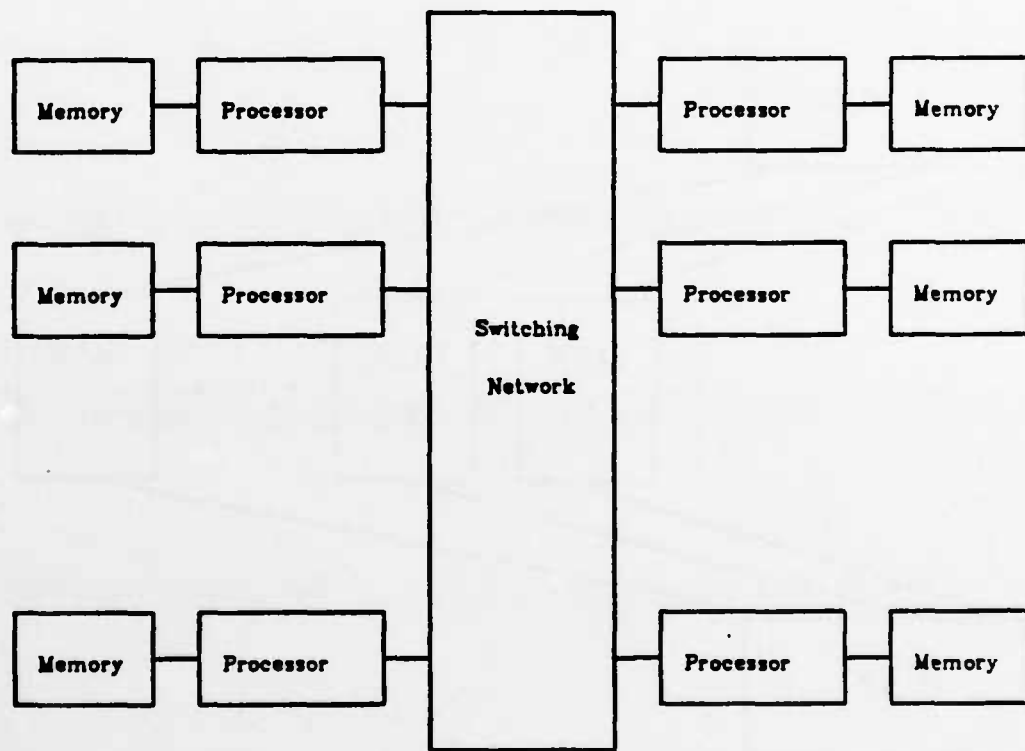


Figure 1.2 A Multiprocessing System

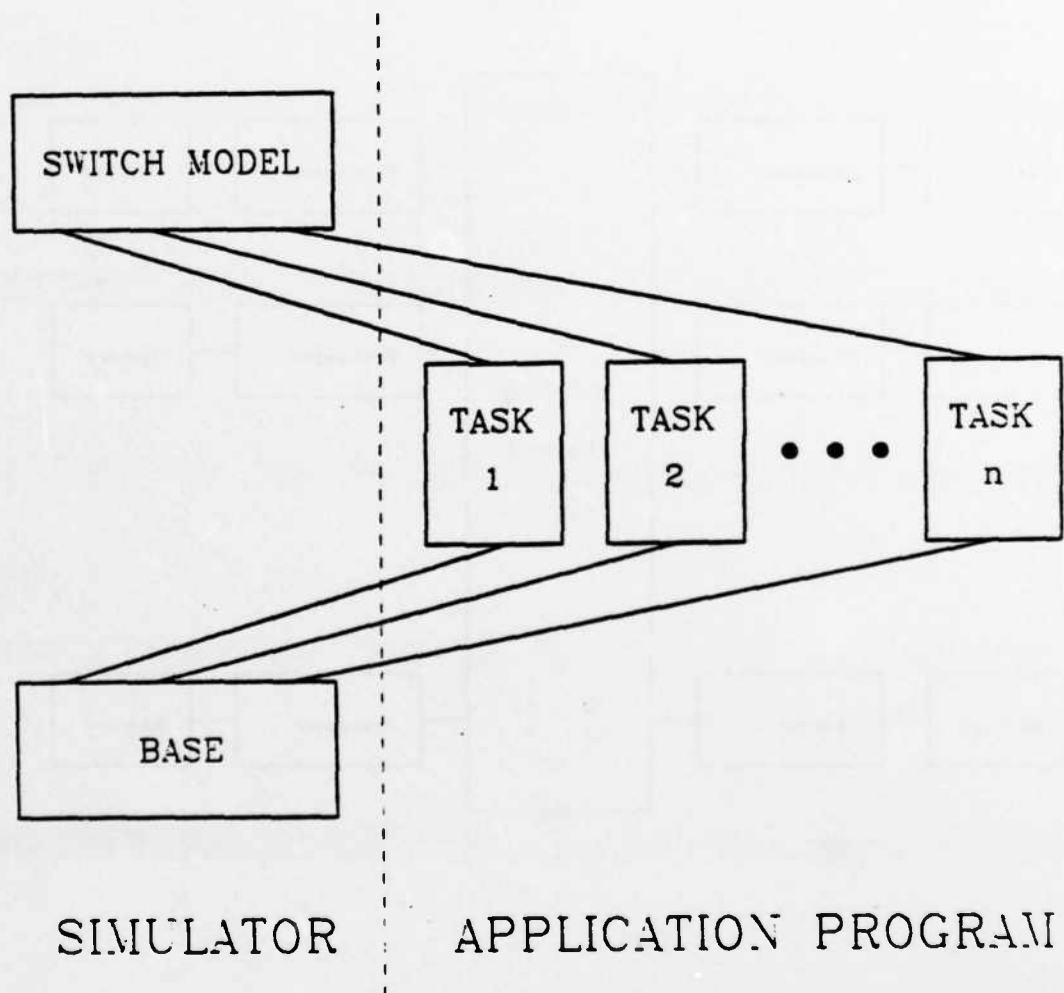


Figure 1.3 Block Diagram of the Simulator, SIMON

## CHAPTER 2

### PARTITION OF LU DECOMPOSITION FOR CONCURRENT PROCESSING

#### 2.1. Introduction

Practically all circuit simulators contain a routine for solving a system of linear equations of the form

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (2.1)$$

where  $\mathbf{A}$  is the coefficient matrix,  $\mathbf{x}$  is the unknown vector of circuit variables and  $\mathbf{b}$  is the excitation vector. In circuit simulation as well as in other engineering applications, the coefficient matrix is very sparse. This inherent sparseness of the matrix must be recognized in order to achieve an efficient solution of the unknown vector  $\mathbf{x}$ .

There are basically two methods of solving (2.1). The first method is called the iterative method, the most common iterative method being the Gauss-Seidal method. This technique involves the initial guess of the solution, then updating the solution vector using the previous iterated solution until convergence is obtained. A detail analysis of the convergence properties and roundoff error can be found in [7]. The second method is called direct method, and solves (2.1) by multiplying both sides of the matrix equation by  $\mathbf{A}^{-1}$  to yield

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b} \quad (2.2)$$

provided that  $\mathbf{A}^{-1}$  exists. The next step could be directly invert  $\mathbf{A}$  numerically. However, an operation count which counts the number of multiplications and divisions required to find the inverse of  $\mathbf{A}$  shows that it requires three times the computational effort of the other direct methods. So the method of finding the inverse of  $\mathbf{A}$  in solving (2.1) is seldom used in the simulation programs. There are

two equivalent methods of direct elimination which do not require finding the inverse of  $A$ , Gaussian elimination and LU decomposition. They are computationally equivalent in the sense that they require the same number of operations. However for multiple input vectors  $b$ , LU decomposition is done only once and the required solutions can be found subsequently by forward and backward substitutions. This method will be discussed in more detail in the next section. For Gaussian elimination, the solution process has to be repeated for each input vector  $b$ . Hence LU decomposition is the sensible choice.

## 2.2. Doolittle Algorithm for LU Decomposition

LU decomposition factors or decomposes the original coefficient matrix  $A$  into the product of a lower triangular matrix  $L$  and an upper triangular matrix  $U$ . Either the diagonal terms of  $L$  or  $U$  are set to unity depending on how the decomposition is carried out. There are several equivalent methods for LU decomposition, that is finding the  $L$  and  $U$ . In the discussion that follows, we present the Doolittle algorithm[13]. The diagonal elements of  $L$  in this case are set to unity.

By substituting  $A = LU$  into the original equation (2.1), we obtain

$$Ax = LUx = b \quad (2.3)$$

Let  $y = Ux$ , then the original system of linear equations is equivalent to two derived systems of linear equations

$$Ly = b \quad (2.4)$$

$$Ux = y \quad (2.5)$$

The method for finding the  $L$  and  $U$  matrices will be discussed in section 2.2.2.

### 2.2.1. Forward and Backward Substitutions

After the factorization, the solution vector  $b$  can be obtained by two successive substitutions: the forward substitution followed by the backward substitu-



tion. From (2.4),  $y$  can be solved easily because  $L$  is a lower triangular matrix by the following recursive relation

$$y_1 = b_1 ; \quad (2.6)$$

$$y_j = b_j - \sum_{i=1}^{j-1} l_{ji} y_i ; \quad j = 2, 3, \dots, N$$

The above process is called forward substitution. After solving for  $y$  and from (2.5),  $x$  can also be solved easily by the following recursive relation

$$x_N = y_N ; \quad (2.7)$$

$$x_j = (y_j - \sum_{i=j+1}^N u_{ji} x_i) / u_{jj} ; \quad j = N-1, N-2, \dots, 1$$

This is called backward substitution. Note that in both forward and backward substitutions, neither the inverse of  $L$  nor the inverse of  $U$  is necessary in finding the solution of (2.1).

### 2.2.2. Doolittle Algorithm

The Doolittle algorithm discussed here consists of two alternating steps. The first is the dividing step and it is followed by the updating step. At each computation step of the algorithm, the dividing step is performed on the elements in the pivoting column and the updating step is performed on all the element whose row index and column index are greater than the corresponding pivot row index and the pivot column index. The detail of the algorithm is as follows:

Given a matrix of size  $N$

*Initialization* :  $k=1$  sub-matrix = original matrix ;

*Step 1* :

In column  $k$  of the sub-matrix, divide all nonzero elements in this column by  $a_{kk}$ . (dividing step)

$$a_{jk} = a_{jk} / a_{kk} ; \quad j = k+1, k+2, \dots, N \quad (2.8)$$

*Step 2* :

For each nonzero element  $a_{ij}$  in the sub-matrix obtained by deleting the first  $k$  rows and  $k$  columns, subtract from it the product of  $a_{ik}$  and  $a_{kj}$ . (updating step)

$$a_{ij} = a_{ij} - a_{ik} * a_{kj} ; i, j = k+1, k+2, \dots, N \quad (2.9)$$

*Step 3 :*

$k = k + 1$ ; If  $k = N$  stop. Otherwise go to *Step 1*.

The method described above gives an in-place LU factorization. The elements  $L_{ij}$  in  $L$  are obtained from the resultant matrix  $A$  as follows :

$$L_{ij} = \begin{cases} 0 & \text{if } i > j ; \\ 1 & \text{if } i = j ; \\ a_{ij} & \text{if } i < j ; \end{cases}$$

and the elements in  $U$  is obtained as follows :

$$u_{ij} = \begin{cases} a_{ij} & \text{if } i > j ; \\ 0 & \text{if } i \leq j ; \end{cases}$$

Before leaving this algorithm and giving a simple example, a few comments are worth mentioning when the given matrix is very sparse. The first comment is that the trivial operation of multiplying a number by zero should be avoided. In updating an element  $a_{ij}$  in the sub-matrix ( see equation (2.9)), if either  $a_{ik}$  or  $a_{kj}$  is zero, then this updating operation is not necessary.

The second comment relates to the generation of fill-in elements. A fill-in element is an element which originally has a value equal to zero but assumes a nonzero value during the LU decomposition. This happens in the updating step when  $a_{ij}$  is zero and neither  $a_{ik}$  nor  $a_{kj}$  is zero. Because of the propagation nature of the fill-in elements, i.e. fill-in elements generate more fill-in elements, it decreases the sparseness of the original coefficient matrix. Various reordering algorithms such as Markowitz, Berry and Nahkla[14] are implemented in the simulation programs to minimize the generation of these fill-in elements. Throughout this dissertation, we assume that this reordering has been done

before performing the Doolittle algorithm.

Another practical aspect of the Doolittle algorithm is the pivoting for accuracy. Pivoting is the interchanging of rows and columns so that the element with the largest magnitude is placed in the pivot position along the diagonal in the matrix. This will reduce the roundoff error introduced in the dividing step. This will not be carried out when the LU decomposition is performed. We will address this point at the end of section 2.3.

In order to understand the Doolittle algorithm in more detail, the following simple example will serve this purpose. Let

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 6 & 6 & 10 \\ 4 & 11 & 7 \end{bmatrix}$$

Steps 1 and 2 with  $k = 1$  produce the following matrix

$$\begin{bmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \\ 2 & 9 & 1 \end{bmatrix}$$

Then steps 1 and 2 with  $k = 2$  give the final matrix

$$\begin{bmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \\ 2 & 3 & -2 \end{bmatrix}$$

and the corresponding two factors can be written down immediately

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 3 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 1 & 3 \\ 0 & 3 & 1 \\ 0 & 0 & -2 \end{bmatrix}$$

### 2.3. Code Generation

Since the routine to solve the system of linear equations is called many times during the whole time interval of the simulation, the idea of generating machine code for solving the system of linear equations has been implemented in SPICE[1]. For dc and transient analysis where the system of equations is real, a nonlooping set of machine code instructions specific to a particular circuit is

generated for fast execution compared to a FORTRAN routine. Since much of the cpu time is spent in addressing when executing a loop, it is beneficial to generate non-looping code in order to gain execution speed. The decrease in cpu time is well worth the increase in memory for storing the additional code. This idea is well suited to the Doolittle algorithm because it is basically a loop.

Another advantage is that the structure of the coefficient matrix for a given circuit is fixed. By structure we mean the locations of the nonzero elements in the sparse coefficient matrix. Hence the operations (divide and update) on the nonzero elements are pre-determined for a given circuit throughout the LU decomposition. In other words, once the sequential code of the divide and update operations of the LU decomposition is generated, it can be used for the entire circuit simulation involving many LU decompositions. This is called symbolic LU factorization. In the following sections, the code generator is described and an example will be given to illustrate this concept.

### 2.3.1. The Code Generator

A computer program, the code generator, was written to generate the sequential instructions of divide and update operations to carry out the LU decomposition. The code generated was a high level language, which was C[15] in this case. The code generator essentially goes through the Doolittle algorithm for a given sparse matrix with the two dimensional linked list data structure as in[1], puts down the necessary operations, adds fill-in elements into the linked list until we finish the entire decomposition process. One of the advantages offered in C is the easy manipulation of files and its standard I/O capabilities. A file is open and all the sequential operations are stored in that file. Each line of code in the file is identified by a number. These numbers will facilitate the detection of parallelism among the operations, which will be discussed in the next section. In the actual implementation, these numbers are the indices to an

array of pointers pointing to the corresponding operations.

### 2.3.2. Simple Example of the Code Generated

With the description of the concept of code generation in the previous section, perhaps a simple example of a file containing the code generated from the code generator will make the idea more clear and concrete. Suppose we are given the following "sparse" matrix,  $A = [a_{ij}]$

$$A = \begin{bmatrix} x & & x & & x \\ x & x & & & x \\ & & x & & x \\ & x & & x & x \\ x & & & & x \\ & & x & & x & x \end{bmatrix}$$

where  $x$ s are the locations of the nonzero elements. After the Doolittle algorithm, the resultant matrix will be

$$A = \begin{bmatrix} x & & x & & x \\ x & x & \emptyset & x & \emptyset \\ & & x & & x \\ & x & & x & x & \emptyset \\ x & & \emptyset & x & \emptyset \\ & & x & & x & x \end{bmatrix}$$

where the  $\emptyset$ s are the locations of the fill-in elements. The code generated for the above example is shown in Figure 2.1. The  $ax_y$  symbol in the code denotes the element in row  $x$  and column  $y$ . Note that the code generated does not contain any loops and hence no addressing is necessary. The code represents a sequential computations only on the nonzero elements of the matrix. So by compiling the above code and running it on a single processor, we will obtain the in-place LU decomposition of the given matrix.

Let us return to the problem of pivoting for accuracy. Pivoting is difficult when using code generation since some of the variable names  $ax_y$  will be changed when the swapping of rows and columns is done in pivoting. If pivoting is necessary, then the reordering of the pivots is performed and the whole process

of code generation has to be repeated.

## 2.4. Graph Model for LU Decomposition

The sequence of instructions of divide and update operations generated by the code generator represents the in-place LU decomposition process. In this section, we will represent the LU decomposition process by a directed graph. Based on this directed graph, scheduling algorithms can be applied to assign the nodes ( which represent the divide or update operations ) to different processors for concurrent execution.

### Definition

A *directed graph*  $G = (N, E, <)$  is a graph which consists of a set of nodes  $N$  and a set of edges  $E$ . The precedence constraints between a pair of nodes  $i$  and  $j$  is denoted by  $i < j$  which means that node  $i$  is the *immediate predecessor* of node  $j$  and node  $j$  is the *immediate successor* of node  $i$ . If nodes  $i$  and  $j$  represent two tasks( processes), then node  $i$  has to be completed before node  $j$  can begin its execution.

In the following sub-sections, we will go through the steps in more detail for obtaining the directed graph from the code generated. Based on this graph, what are the scheduling techniques we can use in order to reduce the execution time of the LU decomposition of a given matrix. Scheduling techniques refer to the problem of assigning nodes in the directed graph to processors for concurrent execution. Then a brief discussion of the extension to the other algorithms and a few simple illustrating examples are given at the end of this chapter.

### 2.4.1. Detection of Precedence Constraints

A careful study of the operations in the code generated reveals that it is not

necessary to execute the code in the order in which the operations are generated. In other words, some operations can be executed simultaneously. The main objective of this sub-section will be devoted to detect which operations can be done in parallel and which operations have to be completed before the other operations can begin their executions. Before we go on, let us define two simple terms. The *input variables* are those variables appearing on the right hand side of the assignment statement of an operation in the code. The *output variable* is the variable appearing on the left hand side of the assignment statement. Notice that in this particular LU decomposition application, there are only two different operations, namely the divide and update operations. In the divide operation, there are two input variables. In the update operation, there are at most three input variables. In both cases, there is only one output variable.

The idea of detecting which operation precedes the other is very simple. If the input variables of an operation are the output variables of some previous operations, then there are precedence constraints between these operations. Otherwise, that particular operation can be initiated immediately.

#### 2.4.2. Implementation of Detection of Precedence Constraints

In order to implement the above detection procedure efficiently, a method having the following two properties should be employed:

- (1) It will identify each output variable uniquely.
- (2) It can search all the output variables as efficiently as possible.

The idea of *hashing*[16] seems to fit the above two requirements and a *hash function*,  $f()$  is needed in this case. In the most simplest terms, a hash function takes a key,  $K$  ( e.g. a string of characters) as input and produces a number called the *hash code*,  $f(K)$  as output. It basically breaks some aspects of the key and use this information as the basis for searching.



There are cases where two distinct keys  $K_i \neq K_j$  hash to the same hash code  $f(K_i) = f(K_j)$ . Such an occurrence is called *collision*. Functions which avoid duplicated hashed values are rare. But some of the sophisticated hash functions have a very low probability of collision. If a collision has occurred, one possible remedy is a technique called separate chaining[16]. It essentially maintains several linked lists. The number of linked lists depends on the largest hash code from these keys. All the keys having the same hash code are linked together. The linked list contains the appropriate information about all these keys

In C, there is a convenient way of storing certain information about an object. It is called *structure* (in Pascal, it is called *record*). A structure can have many *fields* associated with the object. In this particular application, each key has a structure with two fields containing the character strings of the key and a pointer pointing to the next key with the same hash code. The following simple example illustrates this chaining scheme. For example there are five keys;

$$K = AB, CAA, BT, TE, ZXWP$$

having the following respective hash codes:

$$f(K) = 3, 1, 4, 1, 6$$

Note that *CAA* and *TE* have the same hash code. Then the separate chaining scheme is shown in Figure 2.2. In this figure, a pointer pointing to NULL signifies the end of the linked list.

The hash code is the index of a pointer array pointing to these keys. The problem of resolving the collision by this scheme will be discussed in the next sub-section when we explain how to get the LU graph model.

### 2.4.3. The LU Graph Model

In this section, the procedure for obtaining the LU graph model from the code generated is described. It involves the detection of the precedence constraints among the operations in the code. At the end of this section, we will go

through one step of the procedure to illustrate the detection process.

Based on the above discussion of the hashing technique, we can consider each output variable in the code as a key in the separate chaining scheme. The structure of an output variable has a field containing a string of characters of the output variable and a field containing a pointer pointing to the next output variable with the identical hash code. Recall that at the end of section 2.3.1, each operation in the code generated is identified by a number. This number is the index to an array of pointers pointing to that operation. It also gives the position of the operation (as well as the output variable associated with that operation) with respect to the other operations. Besides maintaining the two fields mentioned above in the structure of an output variable, an additional field indicating the latest position of the output variable is needed. The reason is that an output variable can be subjected to several divide and update operations throughout the decomposition. This output variable then will appear at several locations in the code. Thus the latest position of the output variable should be used in order to have the correct precedence constraints among the operations. This will become more clear in the subsequent discussion. The structure of an output variable for the purpose of detecting the precedence constraints is defined below :

```
struct out_var {
    char name[50]; /* character array holding the character
                   string of the output variable */
    struct out_var *ptnxt; /* pointer to next output variable
                           having the same hash code */
    int position; /* most recent position of the output
                  variable in the code */
}
```

By using the above implementations, the detection procedure will be described to illustrate how a graph model for LU decomposition is obtained. Suppose we are at the  $t^{th}$  operation which we will call the current operation in the code, the input variables of the current operation are hashed one at a time

and search the hash codes of the output variables of the previous  $(i - 1)$  operations. If the hash code of the current input variable is equal to one of the hash codes of the previous output variables, then this particular linked list is transversed and compared with the string of characters of the input variable with all the strings of characters of the output variables in that linked list. If there is a match, there exists a precedence relationship between the current operation  $i$  and the previous operation indicated by the position field in the structure of that output variable. This position field gives the latest position that this output variable appears in the code. If there is no match, a collision has occurred and the current operation can be initiated immediately.

After all the input variables have been tested, the output variable of the current operation is hashed. Again the hash code is used to search through the previous  $(i - 1)$  output variables. If it is equal to one of the hash codes of the previous output variables, we transverse that linked list and compare the current input variable with the output variables in that linked list. If there is a match, we then update the latest position of this output variable. If there is no match, a collision has occurred and the current output variable is added to the end of the linked list. If the hash code of the current output variable is not equal to one of the hash codes of the previous output variables to start with, we create a linked list for this current output variable and add it to the pool of the previous  $(i - 1)$  output variables to be searched in the next stage of detection.

The above procedure is repeated for  $i = 1$  until  $i = M$  where  $M$  is the total number of operations in the code. In order to understand the above procedure of detecting the precedence constraints, let us go through one step of the procedure by taking the code generated shown in Figure 2.1. Each operation is identified by a number signifying its position in the code. For example  $a2\_1 = a2\_1 / a1\_1$  is identified as the first operation and  $a5\_1 = a5\_1 / a1\_1$  is identified

as the second operation and so on. Suppose the current operation we are now working on is 13 (  $i = 13$  ), this operation is  $a5\_4 = a5\_4 / a4\_4$ . Its input variables  $a5\_4$  and  $a4\_4$  are hashed. For simplicity, let their corresponding hashed codes are 32 and 27 respectively. ( i.e.  $f(a5\_4) = 32$  and  $f(a4\_4) = 27$  ) Then a search is done on all the hashed codes of the previous twelve output variables ( since there are twelve operations before the current one ) to see whether these hashed codes match with the hashed codes of the current input variables. In this example, two matches are found. One match corresponds to the input variable,  $a5\_4$  which is the output variable of the fifth operation in the code ( this output variable should have 32 as its hashed code ). Another match corresponds to the input variable,  $a4\_4$  which is the output variable of the eighth operation ( this output variable should have 27 as its hashed code ). Hence a precedence constraint exists between the 5<sup>th</sup> operation and the 13<sup>th</sup> operation and also between the 8<sup>th</sup> operation and the 13<sup>th</sup> operation. Then the output variable  $a5\_4$  of the current output variable is hashed. This hashed code is compared with the hashed codes of the previous twelve operations. A match is found because this current output variable,  $a5\_4$  also appears as the output variable of the fifth operation in the code. In this case, the position field in the data structure of this output variable is updated to 13 from its original value of 5. This updating is necessary. Because in the later steps of detection, if an input variable of an operation happens to be  $a5\_4$ , then the 13<sup>th</sup> operation ( which is the most recent operation in which  $a5\_4$  is the output variable ) is identified as the predecessor rather than the 5<sup>th</sup> operation is taken incorrectly.

An example of the graph model for decomposition is shown in Figure 2.3. This graph results from the example in section 2.3.2 by running the code generated through the detection of precedence constraints process. Each node in the graph represents either an update operation or a divide operation. The

number associated with each node represents position in the code. That is node  $i$  represents the  $i^{\text{th}}$  operation from the beginning of the code.

#### 2.4.4. Data Structure for LU Task Graph

In order to manipulate the LU task graph by a computer, a convenient way of representing the precedence relationships among these nodes is necessary. There are many ways of representing a directed graph in a computer. Here we choose a matrix representation called the connectivity matrix representation. Each element  $c_{ij}$  of the connectivity matrix  $C$  is defined as follows :

$$c_{ij} = \begin{cases} 1 & \text{if node } i \text{ is the immediate predecessor of node } j ; \\ \text{NULL} & \text{otherwise;} \end{cases}$$

Note that in the LU task graph, there are at most three immediate predecessors for a node because there are at most three input variables for an update operation and there are only two input variables for a divide operation. Hence the connectivity matrix is extremely sparse. There are no more than three 1's per row. A natural choice for the data structure of the connectivity matrix is the linked list. Since the manipulations on the LU task graph involve the deleting the rows and columns in the connectivity matrix, an easy access to any of the nonzero element from the deleting row and column in the matrix will be a very efficient implementation. Hence a symmetrically double linked list is employed as the data structure for the connectivity matrix. Each nonzero element in the matrix is a structure in  $C$  defined as below :

```
struct element { int irow, jcol;
    struct element *pt_top;
    struct element *pt_right;
    struct element *pt_down;
    struct element *pt_left;
};
```

This structure has six fields. Two of these contain the row index (*irow*) and column (*jcol*). The other four are pointers pointing to the nonzero element to its right (*\*pt\_right*), left (*\*pt\_left*), above (*\*pt\_top*) and below (*\*pt\_down*). With this data structure, we can reach any nonzero element from the deleting row and column very conveniently. A schematic diagram of this symmetrically double linked list data structure is shown in Figure 2.4

## 2.5. Processor Scheduling Techniques

Processor scheduling implies that tasks (i.e. code segments in a program or nodes in the LU task graph) are to be assigned to a particular processor for execution in a particular order. At a certain time step, there may be more than one task ready for execution. Hence it is necessary to have a representation which conveniently represents the relationship among these tasks. A directed graph or precedence graph representation such as the LU task graph is probably the most popular representation in the scheduling literature. The nodes in the graph represent the independent operations which are related to each other in time. The three assumptions cited in the theory of deterministic scheduling are :

- (1) The directed graph is *acyclic*. That is there are no loops or cycles in the graph. The presence of a loop makes a scheduling before execution time impossible since the conditional which controls the number of iterations cannot be known until execution time.
- (2) The execution time of each node is known in advance. That is we are not dealing with stochastic scheduling in which the execution times are random variables.
- (3) Interruption before task completion is not allowed. This is called the nonpreemptive scheduling technique. In some cases where preemptive technique (interruption of a task is allowed) can generate better schedules

than nonpreemptive techniques due to the efficient allocation of processors computing resources. However if preemption occurs frequently, the overhead of system interrupt processing and the additional memory required to store the state of the interrupted task may outweigh the benefits of preemptive scheduling.

### 2.5.1. Performance Criteria of a Schedule and Efficiency of an Algorithm

To evaluate the effectiveness of the schedules, the most often quoted measures are:

- (1) Minimize the completion time.
- (2) Minimize the number of processors needed.

Presently the quest for computing speed seems of greater concern than the cost of hardware. Hence the first criterion is far more important than the second one. Most of the effort is directed towards minimizing the completion time of a given job.

The running time of an algorithm to locate a processor schedule is also very important. For the purpose of comparison, we call an algorithm *efficient* if it requires computation time which is bounded in the size of its input by some polynomial. An *inefficient* algorithm is the one which requires an enumeration of all possible solutions before the optimal schedule can be chosen. The running times of these algorithms are exponential to the size of its input. For most of the scheduling problems stated in their generalities, very few have efficient algorithms to obtain the optimal schedules. This family of intractable problems are called *NP complete* problems.

There are various scheduling algorithms for different scheduling environments. Under some very restricted constraints, optimal processor schedules can be obtained. An example used here is the level scheduling technique by



Hu[17]. The scheduling environment in Hu's algorithm is that the directed graph is a rooted tree and the execution times on the nodes of the tree-structured graph are equal. The details of Hu's level scheduling algorithm will be discussed in the next chapter.

Under some other unrestricted environments such as arbitrary graph structures, two or more processors, unequal task execution times, developing heuristic scheduling algorithms with polynomial bounded running time which give suboptimal schedules is a feasible approach. Five heuristic scheduling algorithms are studied by Adam et al.[18] Two of the heuristics are based on the concept of the *level* of a node, one is assigning tasks randomly to processors and the other two are based on the concept of *co-level*.

In the next three chapters, the level concept of a node is used extensively in the heuristic scheduling algorithm when there is communication delay in the switching network. Hence it will be appropriate to define the level associated with a node, which is the sum of all the execution times of the nodes on the longest path from the node to the terminal node. The *co-level* of a node is measured in the same manner as the level, except that the length of the path is computed from the entry node rather than from the terminal node. Extensive simulation results have shown that the heuristics having priority assigned according to level (i.e. the larger the level, the higher the priority) perform better than the others. These heuristics are referred as longest-path scheduling. The level of performance of these heuristics has been reported by Adam et al to be within 4.4% of optimal. This near-optimality has also been observed by other researchers such as in[19] and[20]. It demonstrates that this longest-path scheduling is indeed an excellent candidate for obtaining processor schedules under the above stated environments.

There are other scheduling algorithms of interest. Ramamoorthy, Chandy

and Gonzalez[21] developed algorithms to determine the minimum number of processors required to process an arbitrary graph in minimum time and to determine the minimum time to process a graph given the number of processors.

## 2.6. Extension to Other Algorithms

The idea of obtaining the graph model by exploiting the maximum parallelism in the algorithms can be extended to many other areas such as digital signaling processing. In this section, three parallel algorithms are presented. These three algorithms are the inner product of two vectors, the linear convolution of two sequences and the FFT. For each algorithm, the graph model or data dependency graph is illustrated to show the communication pattern for all the operations.

### 2.6.1. The Inner Product

Consider the problem of computing the inner product of two vectors  $a = (a_1, a_2, \dots, a_N)$  and  $b = (b_1, b_2, \dots, b_N)$ . This can be performed in a program loop as follows:

```
for(i=1; i≤N; i++)
    x = x + ai * bi ;
```

where  $x$  is initially set to zero. The final value of  $x$  will be the inner product of  $a$  and  $b$ . We can expand the expression inside the loop by substituting the right hand side into itself as follow:

```
x = a1 * b1 ;
x = a1 * b1 + a2 * b2 ;
x = a1 * b1 + a2 * b2 + a3 * b3 ;
.
.
.
x = a1 * b1 + a2 * b2 + ..... + aN * bN ;
```

The last step of the above  $N$  iterations reveals that considerable parallelism exists in computing the inner product of two given vectors. For the sake of simplicity in explanation, let us assume  $N = 4$  and the code generated is shown below:

```

 $x_1 = a_1 * b_1;$ 
 $x_2 = a_2 * b_2;$ 
 $x_3 = a_3 * b_3;$ 
 $x_4 = a_4 * b_4;$ 
 $x_5 = x_1 + x_2;$ 
 $x_6 = x_3 + x_4;$ 
 $x_7 = x_5 + x_6;$ 

```

It is clear from the above seven steps that the first four steps can be done simultaneously. Then the 5<sup>th</sup> and the 6<sup>th</sup> can be performed concurrently next. Hence the computational graph or the data dependency graph for that particular example is shown in Figure 2.5. In general, if the two given vectors are of size  $N$ , the inner product computation can be mapped into a tree of  $\log_2 N$  levels. The whole computation can be done in  $O(\log_2 N)$  time steps by using  $O(n)$  processors. By using a single processor,  $2N-1$  time steps are required where one time step is either a multiplication or addition.

### 2.8.2. Linear Convolution

In most applications of digital signal processing, we might be interested in implementing a linear convolution of two sequences. An example is the filtering of a sequence such as speech waveform or a radar signal. Let us consider the two  $N$ -point sequences  $h_n$  and  $x_n$  and let  $y_n$  denote their linear convolution:

$$y_n = \sum_{m=0}^{N-1} x_m h_{n-m} \quad (2.12)$$

The sequence  $h_n$  can be treated as the given unit-sample response of a linear time invariant system and  $x_n$  is a sequence of sampling values of a signal which we want to perform some filtering, and  $y_n$  is the output sequence.

If (2.12) is expanded, the following output sequence  $y_n$  is obtained :

$$\begin{aligned} y_0 &= x_0 h_0 : \\ y_1 &= x_0 h_1 + x_1 h_0 : \\ y_2 &= x_0 h_2 + x_1 h_1 + x_2 h_0 : \\ y_3 &= x_0 h_3 + x_1 h_2 + x_2 h_1 + x_3 h_0 : \\ &\vdots \end{aligned}$$

Given all the  $h_i$ , we can see immediately that the multiplication of  $x_0$  with  $h_i$  can be done in parallel once we know  $x_0$ . Similarly if we know  $x_1$ , the multiplication of  $x_1$  with  $h_i$  can be performed simultaneously etc. The data dependency graph for the case of  $N = 4$  is shown in Figure 2.6 In general, if the two given sequences are of length  $N$ , their linear convolution can be computed in  $O(2N-1)$  time steps using  $O(n)$  processors. By using a single processor,  $2 \sum_{k=1}^{N-1} (2k-1) + (2N-1)$  time steps are required. Again one time step is either a multiplication or an addition.

### 2.6.3. Fast Fourier Transform

There are many situations such as in speech recognition and in radar systems where digital signal processing involves the analysis of the spectrum of the incoming signal. The discrete Fourier transform (DFT) is the central computation in these spectral analysis problems. The set of algorithms known as fast Fourier transform (FFT) are the fast implementation of DFT. These FFT algorithms can sometimes improve by a factor of 100 or more over the direct evaluation of the DFT.

The DFT of a finite duration sequence  $x(n)$ ,  $0 \leq n \leq N-1$ , is defined as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{nk} \quad k=0,1,\dots,N-1 \quad (2.13)$$

where  $W = e^{-j(2\pi/N)}$ . The idea behind the FFT is to break the original  $N$ -point sequence into two  $(N/2)$ -point sequences which in turn can be further divided into four  $(N/4)$ -point sequences etc. This process can be iterated until we have  $(N/2)$  2-point sequences. This is the radix 2 FFT. The original  $N$ -point DFT is obtained by combining these 2-point DFTs with the appropriate complex scaling factors. A detail discussion of the FFT can be found in [22].

The advantage of implementing FFT on a multiprocessor system lies on the fact that the DFT of the original sequence can be decomposed into many stages of smaller sequences of DFTs. By performing these DFTs of smaller sequences in parallel using many processors, a speedup of  $O(n)$  will be expected. Figure 2.7 shows the data dependency graph for an eight-point DFT constructed from four two-point DFTs.

## 2.7. Conclusion

In this chapter, the Doolittle algorithm for LU decomposition is described. A sequence of operations is generated by a code generator representing the LU decomposition process. Based on these operations, the precedence constraints among these operations are detected and the corresponding LU task graph is obtained. Different scheduling algorithms are discussed and Hu's level scheduling algorithm seems attractive for obtaining a schedule for the LU task graph. In the next chapter, Hu's level scheduling algorithm is applied to several coefficient matrices from some benchmark circuits in SPICE[1]. The idea of generating a graph model for computation to exploit the maximum parallelism has been extended to the inner product computation, linear convolution and FFT. This demonstrates the feasibility of applying this approach to many algorithms.

```

#include <stdio.h>
#include "decl.h"
CODE()
{
    int td1, td2;

    td1 = ugetclk() ;
    a2_1 = a2_1 / a1_1 ;
    a5_1 = a5_1 / a1_1 ;
    a2_4 = -a2_1 * a1_4 ;
    a2_6 = -a2_1 * a1_6 ;
    a5_4 = -a5_1 * a1_4 ;
    a5_6 = -a5_1 * a1_6 ;
    a4_2 = a4_2 / a2_2 ;
    a4_4 = a4_4 - a4_2 * a2_4 ;
    a4_5 = a4_5 - a4_2 * a2_5 ;
    a4_6 = -a4_2 * a2_6 ;
    a6_3 = a6_3 / a3_3 ;
    a6_6 = a6_6 - a6_3 * a3_6 ;
    a5_4 = a5_4 / a4_4 ;
    a5_5 = a5_5 - a5_4 * a4_5 ;
    a5_6 = a5_6 - a5_4 * a4_6 ;
    a6_5 = a6_5 / a5_5 ;
    a6_6 = a6_6 - a6_5 * a5_6 ;
    td2 = ugetclk() ;

    printf("Result For Single Processor:\n") ;
    printf("Time to Run The Code Is %d Microseconds.\n", td2-td1-2) ;
}

```

Figure 2.1 A Simple Example of the Code Generated

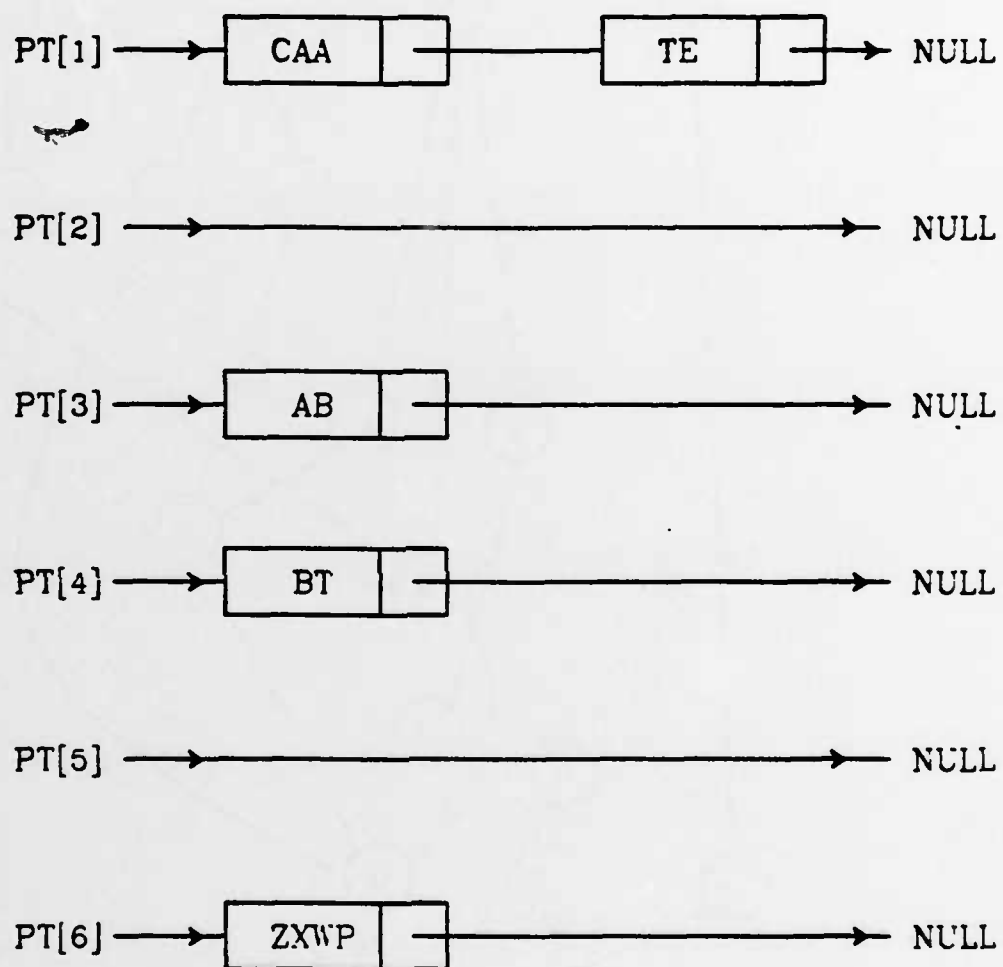


Figure 2.2 An Example of the Separate Chaining Scheme to Handle Collision



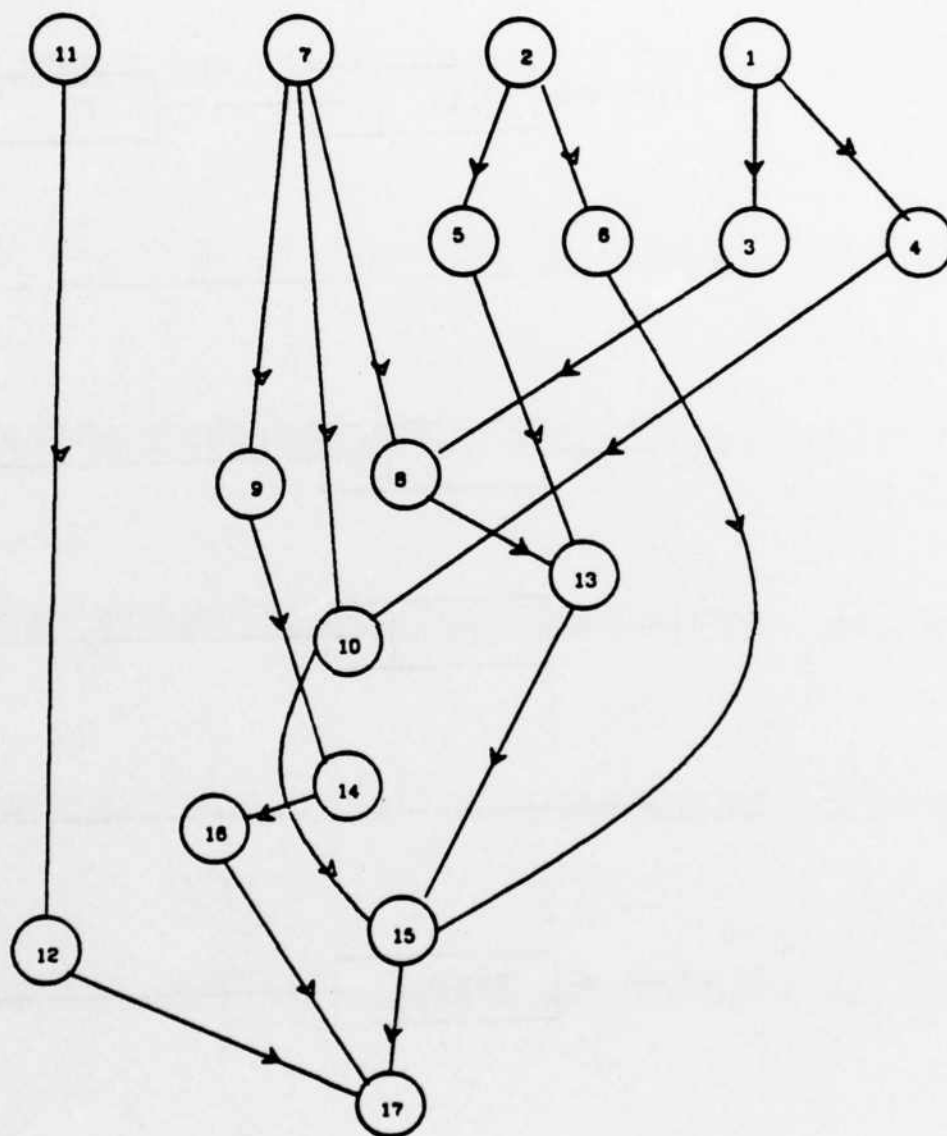
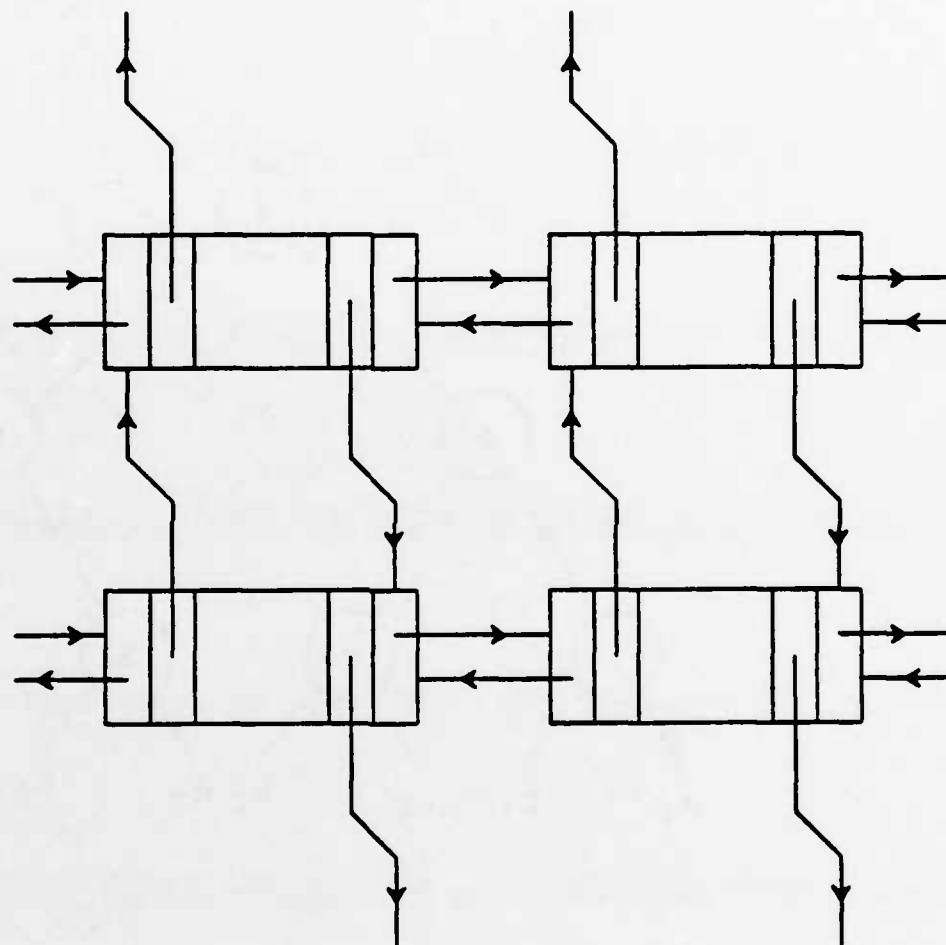


Figure 2.3 An Example of Graph Model for LU Decomposition



**Figure 2.4** Symmetrically double linked list for the Element Data Structure in the Connectivity Matrix

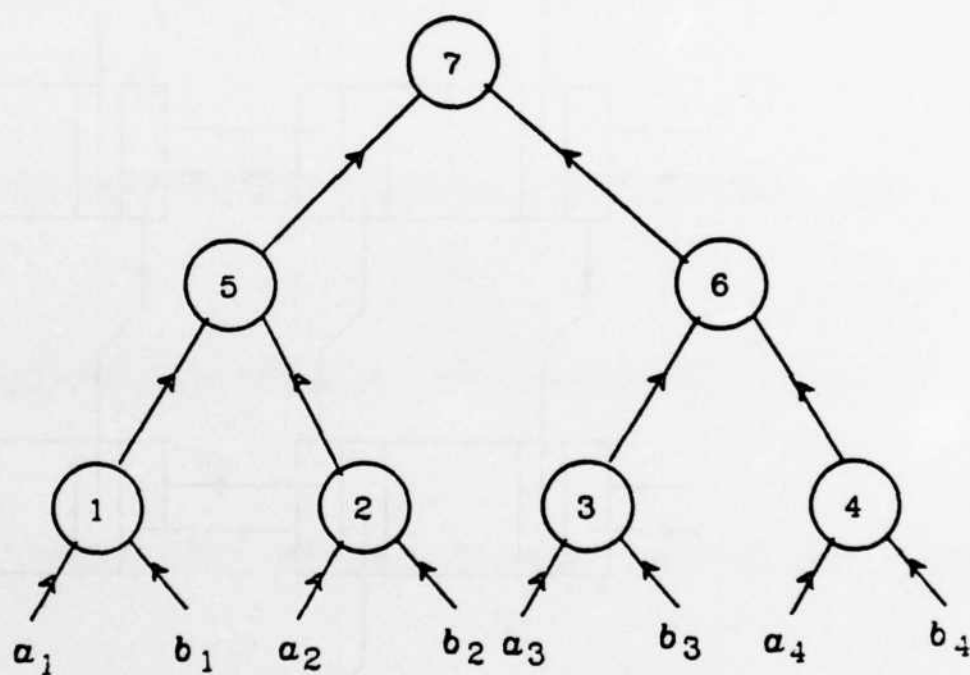


Figure 2.5 The Data Dependency Graph for the Inner Product Example

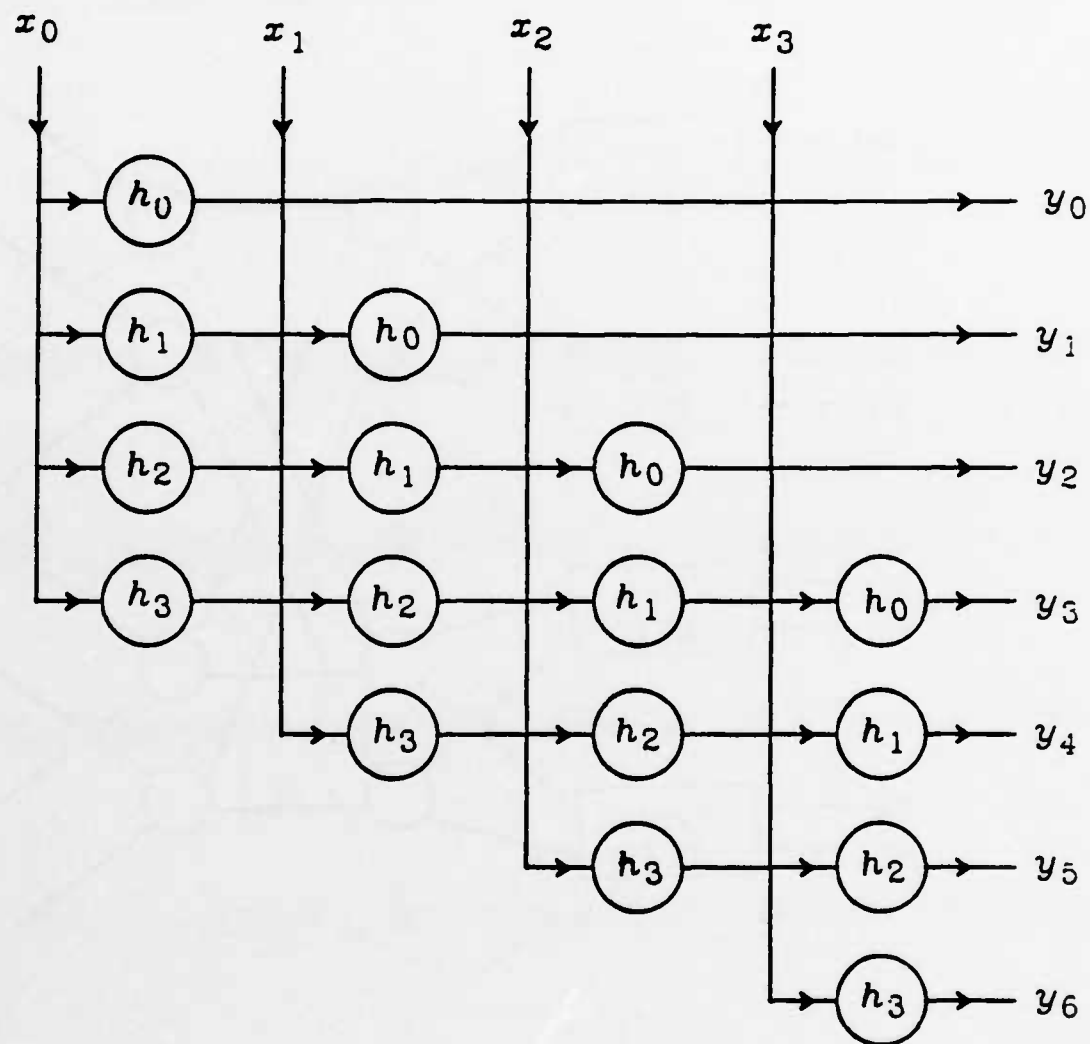


Figure 2.6 The Data Dependency Graph for the Linear Convolution Example

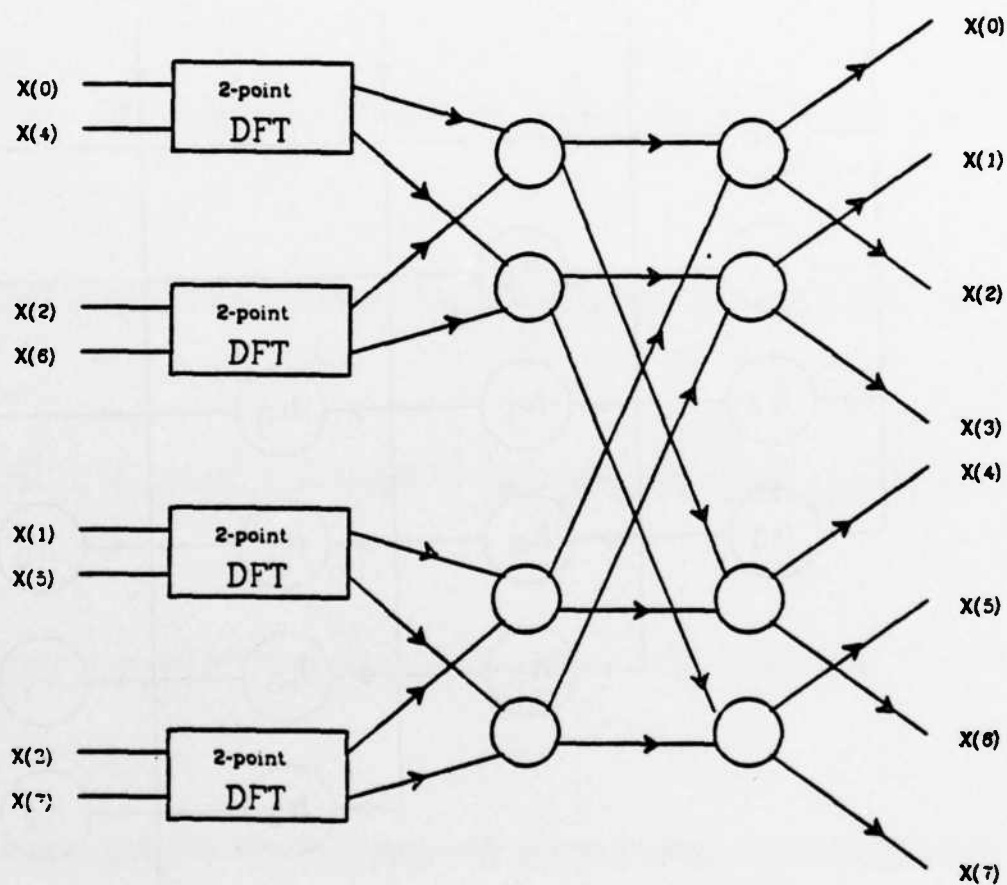


Figure 2.7 The Data Dependency Graph for the FFT Example

## CHAPTER 3

### SCHEDULING ALGORITHMS WITHOUT COMMUNICATION DELAY CONSIDERATION

#### 3.1. Introduction

In chapter two, a detailed procedure for obtaining the graph model representing the LU decomposition of a matrix is described. Based on this LU graph model, existing scheduling techniques can be used to assign the nodes which represent either update or divide operations to processors for concurrent execution. A brief survey on the processor scheduling techniques was summarized in the previous chapter.

In this chapter, a discussion of the list schedules obtained from a class of list-scheduling algorithms will be given. A comparison of these list scheduling algorithms, based on the studies by Adam[18], is made in order to show the near-optimality of the longest path scheduling technique using the concept of *level* associated with a node in the task graph. In the latter part of the chapter, the most often cited scheduling algorithm in deterministic processor scheduling theory, the Hu's level scheduling algorithm[17], is described. The performance of this scheduling technique when applied to several circuit matrices obtained from some benchmark circuits in SPICE[1] is presented. The performance is evaluated in two ways. First, it is measured when there is no delay between communicating processors, and second it is measured when there is a constant delay between processors exchanging data. In the next few sections, the precise meaning of performance in the context of multiprocessing and the communication delay in the multiprocessor system will be discussed. At the end of the chapter, simulation results are presented with a discussion on the impact of the

delay on the speedup performance of Hu's level scheduling algorithm.

### 3.2. List Schedules

In its simplest terms, scheduling is the problem of determining which task ( node ) a given processor should execute at each timestep. List schedules are the most common schedules and are obtained from the list-scheduling algorithms. These schedules are a class of implementable schedules in which each task is assigned a numerical value which is its priority and these tasks are arranged in a list of decreasing order of magnitudes of the priorities. When a processor is available, the list is scanned for an executable task in the list in the order of decreasing priority. If there is more than one executable tasks with the same priority, one task is chosen at random. The way of assigning priorities to the nodes is different for different scheduling algorithms, resulting in different schedules. In this section, a few list-scheduling algorithms are discussed, and in the next section, Hu's level scheduling algorithm ( which can be considered a list-scheduling algorithm ) is discussed.

These list-scheduling algorithms have been studied extensive in[18] . They are based on the notions of *level* and *co-level* associated with a node. These two terms have been defined in chapter two. The scheduling environment is unrestricted in the sense that the precedence constraints between tasks and the execution times of the tasks are arbitrary. These list-scheduling algorithms are :

#### (1) Highest Levels First With Estimated Times

The list schedules generated are obtained from a list in which all tasks are assigned with priorities according to their levels. The larger the level, the higher the priority. This is basically Hu's level scheduling algorithm.

#### (2) Highest Levels First With No Estimated Times

The priority of each task is also its level. However the level is computed



under the assumption that all the tasks have the same execution time. This is useful when no estimates for the task execution times are available. For example, it may be possible to partition a program into modules and yet there are no estimates for the execution time of each module due to the conditional branches etc. In this case, this list-scheduling algorithm may provide satisfactory schedules.

(3) Random

The tasks are assigned priorities randomly.

(4) Smallest Co-levels First With Estimated Times

The priority of a task is the negative of its co-level. In other words, the smaller the co-level, the higher the priority.

(5) Smallest Co-levels First With No Estimated Times

This is the same as (4) above except the co-levels are computed under the assumption that all tasks have the same execution time.

Extensive testing has been done on the performance ( measured by the completion time of the task graph ) on graphs generated randomly. Results have shown that the " Highest Levels First With Estimated Times " scheduling algorithm performs, most of the time, better than the other four list-scheduling algorithms ( see[18] ). This particular scheduling algorithm, which is essentially Hu's level scheduling algorithm deserves more examination and it is discussed in more detail in the next section.

### 3.3. Hu's Level Scheduling Algorithm

The Hu's level scheduling algorithm is perhaps the most referenced algorithm in the scheduling literature. This algorithm is simple and efficient. The running time is  $O(N)$  where  $N$  is the number of nodes in the task graph. It is proved in[17] that this algorithm generates optimal schedules for tree-structure

precedence graphs in which all the nodes have the same processing time. It can be applied to an arbitrary structured graph and satisfactory schedules are obtained most of the time. In this section, this algorithm is applied to graphs obtained from the LU decomposition of circuit matrices, and the performance of this scheduling algorithm is presented.

The LU task graphs obtained from the circuit matrices generally have little structure. To be more specific, the precedence constraints governing the order in which nodes are processed in the LU decomposition can be completely arbitrary. The only assumption we are making is that the execution time of the operations ( either update or divide ) is the same. In this unconstrained scheduling environment, obtaining an optimal schedule for processors seems intractable. Extensive studies as described in the last section have shown that heuristic scheduling algorithms employing the concept of *level* give near-optimal schedules. Hu's level scheduling algorithm proves to be perhaps the most appropriate method to use in this situation.

The definition of *level* of a node is given in the last chapter. It represents the longest distance from that node to the terminal node. The distance is the sum of all the executing times of the nodes along that longest path. An example of a directed graph with the level labelled for each node is shown in Figure 3.1. In a certain sense, the level of a node gives the minimum number of time units required to finish the execution of all the nodes with lower level, assuming there are enough processors available.

Based on this, it is natural to assign a higher priority for earlier execution to nodes with higher level. The high level pseudo-code for Hu's algorithm is given below.

```
while ( there are unassigned nodes )
{
```

Among the nodes ready for assignment to processors, pick the ones with the highest level and schedule them to the available processors. If there are more nodes with the highest level than there are available processors, choose arbitrary node.

}

An example schedule for the graph of Figure 3.1 is given in Figure 3.2. These figures representing the schedules for the processors are referred to as Gantt charts.

In the context of multiprocessing, one of the most important performance criteria of a schedule is the speedup ratio achieved. It is defined as:

$$\text{speedup ratio} = \frac{\text{completion time using one processor}}{\text{completion time using } m \text{ processors}}$$

The speedup ratio achieved as a function of the number of processors available is given in Figure 3.3. This example task graph has 128 nodes. A 45-degree line is drawn in the figure representing the ideal case where given  $m$  processors, a speedup ratio of  $m$  is obtained. This can be also viewed as the case in which all the nodes represent independent operations, and do not have any precedence constraints among them. This example reveals that as the number of processors increases, the speedup ratio increases. When the number of available processors is small, the speedup ratio is very close to the ideal case. As the number of processors increases further, the speedup ratio increases more slowly until at a certain point any gain in speedup ratio is not possible. This can be explained by the fact that when more processors are available than necessary, the idle times in the processors will be larger and they are not performing any computation. This is also related to the bounds which will be discussed in the following sections.

### 3.3.1. Minimum Time Required to Finish the Task Graph

Recall that the level of a node is the sum of all the execution times of the nodes along the longest path from that node to the terminal node. In other words, it is the minimum time units required to finish the rest of the task graph from this node. Hence the largest level number in the task graph gives the minimum time required to complete the task graph. This minimum time is denoted by  $T_{\min}$ . In the following two subsections, the bounds on the number of processors required to finish the task graph by  $T_{\min}$  are derived. The derivation is based on [21] and [17].

### 3.3.2. Maximum Number of Processors Required

This bound is important because it allows a more efficient allocation of computing resources. If the number of processors allocated is larger than this bound, the additional processors will not reduce the time required to complete the task graph. Let  $p(i)$  denote the number of nodes with level equal to  $i$ . Let  $Max\_P = \text{Max} \{ p(i) \}$ . Then if  $Max\_P$  processors are available, all the nodes with the same level can be executed at each timestep. Hence the task graph can be completed in a time equal to  $T_{\min}$ .

### 3.3.3. Minimum Number of Processors Required

Since  $p(i)$  denotes the number of nodes with level equal  $i$ , then  $\sum_{i=1}^x p(i)$  is the total number of the nodes with level equal to or less than  $x$ . In order to finish all these nodes in time  $x$ , it requires at least  $\text{INT} \left[ \frac{1}{x} \sum_{i=1}^x p(i) \right] + 1$  processors where  $\text{INT}[x]$  is an integer obtained by taking the integral part of  $x$ . Let  $L$  be the largest level in the task graph. The minimum number of processors required to finish the execution of all the nodes by  $T_{\min}$  is  $\text{INT} \left[ \text{Max}_{s=1, \dots, L} \frac{1}{x} \sum_{i=1}^x p(i) \right] + 1$ .

### 3.3.4. Maximum Achievable Speedup Ratio

Let  $N$  denote the number of nodes in the task graph and  $t_i$  the execution time for node  $i$ . Then it takes  $\sum_{i=1}^N t_i$  time units to complete the task graph by using one processor. Since it requires a minimum of  $T_{\min}$  time units to finish all the nodes, the maximum speedup ratio that can be achieved is given by  $\sum_{i=1}^N t_i / T_{\min}$ .

### 3.4. Impact of Communication Delay on Speedup Performance

Additional examples on the speedup performance of Hu's level scheduling algorithms are shown from Figure 3.4(a) to Figure 3.4(b). The bounds on the number of processors, the number of levels and the maximum achievable speedup ratio are shown on each example task graph.

All these examples show that indeed Hu's level scheduling technique does give quite promising results when the communication delay between processors is ignored. However for a realistic distributed computing system, the issue of communication overhead cannot be overlooked. It will lengthen the time required to complete the task graph and hence will degrade the speedup performance of a multiprocessing system especially when the communication delay is large compared to the node execution time. In the next few sections, the impact of the delay in transmitting the data on this degradation will be discussed.

In many applications, the main purpose of distributed computing is to increase processing speed by parallel execution, reducing the completion time for a given task. However in any distributed processing environment, the original task is divided into many different modules which are assigned to different processors for concurrent execution. In many real applications, some of the modules depend on the results of other modules in order to continue the execu-

tion. This exchange of data reduces the overall processing speed because there is a finite delay before the necessary message arrives at the destination. This communication delay not only increases the time needed to complete a given task, but it also degrades the distributed system by not allocating the computing resources efficiently. The processors sitting idle waiting for data perform no useful function. It is desirable for the scheduling algorithms to minimize these undesirable effects.

None of the schedules described above considers the communication overhead. Stone[23] has proposed a network flow algorithm for multiprocessing scheduling with communication delay between modules. He formulated the scheduling as a commodity flow problem and obtained an assignment by maximizing the flow through the network. This method is not applicable here because there are no precedence constraints between the modules. With these constraints, the order in which the nodes are executed is restricted severely and this complicates the problem considerably.

Before the impact on the speedup performance is presented, the components of the delay will be described in the following section.

#### **3.4.1. Components of Communication Delay**

In identifying the components of the communication delay in a network, a simplifying assumption is made that the data will not go through a series of intermediate processors before it is forwarded to its final destination processor. In other words, a fully connected network topology for the processors is assumed. The fundamental components of delay are as follows:

##### **(1) Propagation Delay**

This is essentially the delay associated with transversing the circuit. It is defined as the ratio of the circuit length to the signal propagation rate. It is

a function of the physical separation between processors.

(2) Transmission Delay

This is the time required for all the bits of data to be put on or fetch from the circuit. It is the time associated with the execution of the put( ) and get( ) functions in the simulator, SIMON as described in chapter one.

(3) Overhead Processing Delay

This is the time needed by the processor to identify which destination processor the data is being forwarded to. It also includes the time required to store the transmitting data in an export fifo as well as the time required to fetch the data from an import fifo.

In this dissertation, the communication delay refers to the propagation delay. The second and third components represent the system overhead which are ignored in the simulation. The primary concern here is the effect of delay on the speedup performance. When a processor is waiting for the necessary data from the other processors in order to continue to continue its computation, it basically idling serving no useful purpose. This is not an efficient allocation of computational resources. Hence scheduling techniques taking into account the communication delay have to be developed. This will be dealt in the next two chapters. In the remainder of this chapter, the effect of the delay on the speedup ratio achieved is presented. Several circuit matrices are used to test the Hu's level scheduling algorithm with constant delay between communicating processors. The assumption of constant delay simplifies the scheduling problem in the following sense. In a realistic multiprocessing system environment, the delay between two processors depends dynamically on the factors such as the traffic pattern in the interconnecting switch, the topology of the interconnect network, the queuing delay etc. These are usually very difficult to measure. If all these factors are taken into account, this represents a completely different



scheduling environment called a *stochastic* scheduling problem.

### 3.4.2. Determination of Completion Time of a LU Task Graph

In this section, the determination of the completion time of a LU task graph based on Hu's schedule is described when a constant delay is assumed in the switching network of a multiprocessing system. Another assumption is that once the result of an operation is generated, it is forwarded *immediately* to the other processors so that these processors can continue their computation. Also the correct order of the arrival of data is assumed in the receiving processor in order to assure the correct execution of the algorithm.

The determination of the completion time of a LU task graph when there is a constant delay between communicating processors is not as simple as in the case when the delay is ignored. In the latter case, the completion time of a given graph can be obtained from the Gantt chart representing the schedule of the processors. For example, if the LU task graph of Figure 3.1 is scheduled to be executed on three processors, the corresponding Gantt chart is shown in Figure 3.2(b). The completion time of this schedule is 7 units. In the case where there is a delay, it is necessary to go through all the operations scheduled to be executed at each timestep, locating the predecessors of each operation, adding the magnitude of the delay to the times at which the predecessors are executed by the processors they are assigned to, and then finding the maximum of these times. This is the time at which the execution of this node will be completed. A more detailed treatment will be given again in chapter four when scheduling techniques taking into account the communication delay are discussed.

However, the following simple example will illustrate the procedure for determining the completion time of a schedule when there is a delay. A segment of a three processor schedule is shown in Figure 3.5 at timestep equal to 32. At this timestep, node *l* is assigned to processor 2. It is necessary to search

through the graph ( the connectivity matrix ) to find out the predecessors of node  $l$ . Suppose they are nodes  $i, j$ , and  $k$ . Then another search of the schedule of each processor to locate these predecessors is necessary. Suppose they are located as shown and the corresponding times at which they are completed are 15, 18 and 17 time units for nodes  $i, j$  and  $k$  respectively. Assume a delay of 20 units in forwarding a message from the sending processor to the receiving processor, since node  $j$  and node  $l$  are residing in the same processor, there is no need to consider the delay of transmitting the result of node  $j$  to node  $l$ . If processors 3 and 1 will forward the results after the execution of node  $i$  and  $k$  immediately to processor 2, the result of node  $i$  will arrive at processor 2 at time equal to 35 while the result of node  $k$  will arrive at time equal to 37. Hence the time at which processor 2 will finish the execution of node  $l$  is 38 ( which is equal to 37, the maximum delay due to the transmission of the results from the predecessors plus 1, execution time of node  $l$  ). Then the same procedure is repeated for node  $m$  and node  $n$  in the same timestep. After all the nodes are completed at this timestep, the nodes in the next timestep are attempted. This procedure continues until the last timestep is reached and the completion time is found.

Having found the completion time, the speedup ratio is obtained by taking the number of nodes in the task graph, which is numerically equal to the completion time if one processor is used assuming one unit execution time for each node, divided by the completion time obtained as described in the last paragraph.

### 3.5. Results and Discussion

The simulation results of the impact of the delay on the speedup performance are shown from Figure 3.6(a) to Figure 3.6(c). These graphs represent the LU decomposition process of a given matrix. The number of nodes in the

graph is the number of operations required to carry out the decomposition to completion. These schedules are obtained by Hu's level scheduling algorithm and different delays are assumed between communicating processors to see the effect on the completion time of these schedules. As seen from the simulation results, the increase in delay decreases the speedup ratio achieved. When it is expected that the communication overhead will degrade the speedup performance unacceptably, some other scheduling algorithms with delay consideration should be employed.

With the communication delay taken into account, the scheduling problems become even more complicated. Without the delay, any available node can be assigned to any available processor without regard to where the predecessors of this node are located. With the communication delay consideration, the nodes to processors assignments at the previous timesteps affect the assignment at the present timestep. Also there seems to be no possible way of predicting how the present assignment will affect the assignments at the future timesteps.

Another possible tradeoff between parallel execution and reduction in communication delay exists in this scheduling problem. Based on where the predecessors of a node are located, the presence of communication delay tends to assign this node to the same processor as its predecessors have been assigned. This tendency to schedule nodes to the same processor will result in serial execution of these nodes. Hence it does not exploit the possible parallelism of these nodes in the task graph. Since the main performance criterion here is to finish the task graph in the shortest possible time, it is sometimes beneficial to execute these nodes serially, especially when large delay exists in the switching network.

### 3.6. Conclusion

In this chapter, a common class of deterministic processor scheduling is presented. Based on extensive simulation results, Hu's level scheduling algorithm seems to be most suitable for scheduling the LU task graph for concurrent execution when there is no delay between communicating processors. As seen from the simulation results, the speedup performance is almost linear with the number of processors when this number is well below the bound on the minimum number of processors given in section 3.3.3.

However, the presence of delay overhead degrades this speedup performance considerably if Hu's level scheduling algorithm is used to obtain the nodes to processors assignment. Hence scheduling techniques taking into account the communication delay have to be employed in this case. The main objective is to obtain a schedule such that a LU task graph can be finished in the shortest possible time. While a complete solution to this problem appears to be inherently intractable, the next two chapters attempt to solve this problem through the use of heuristics.

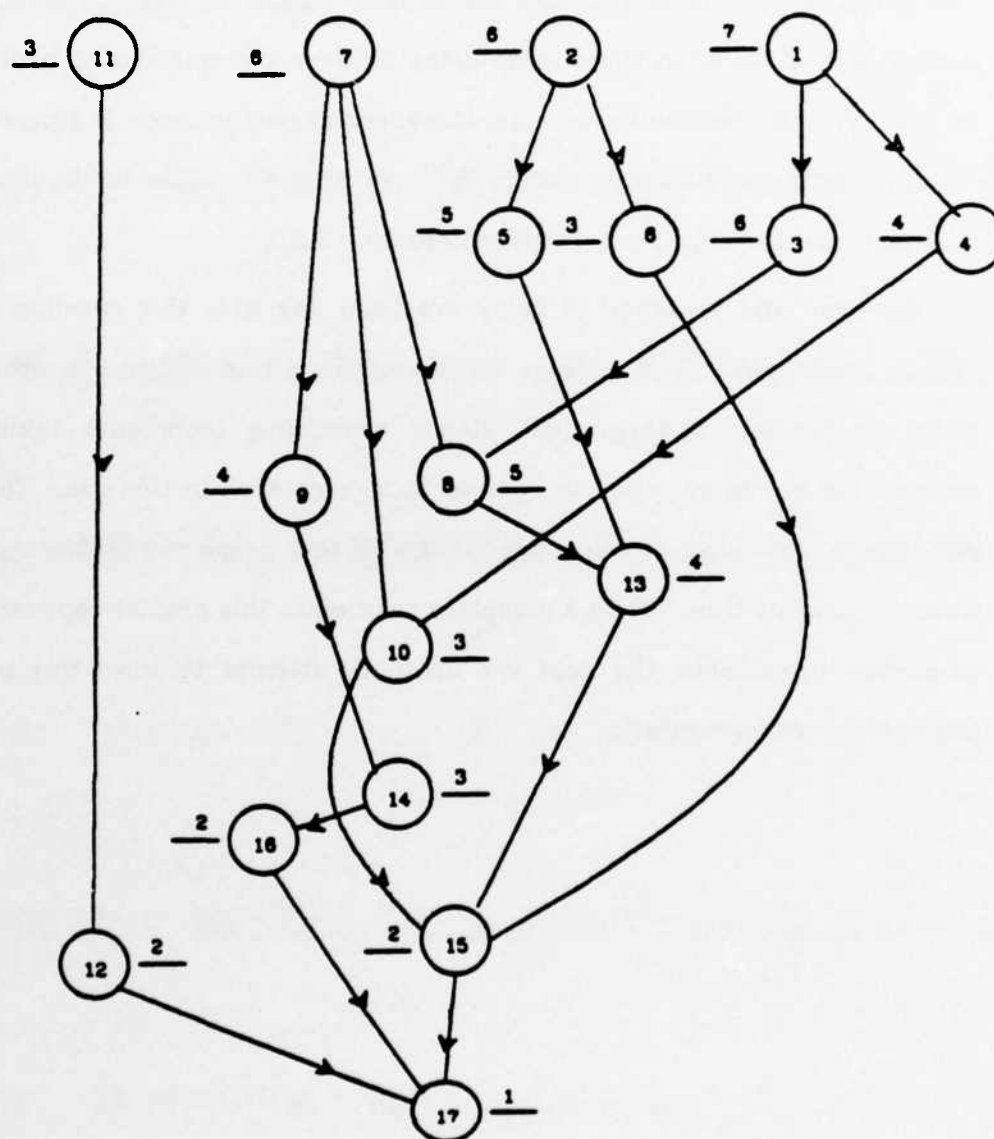
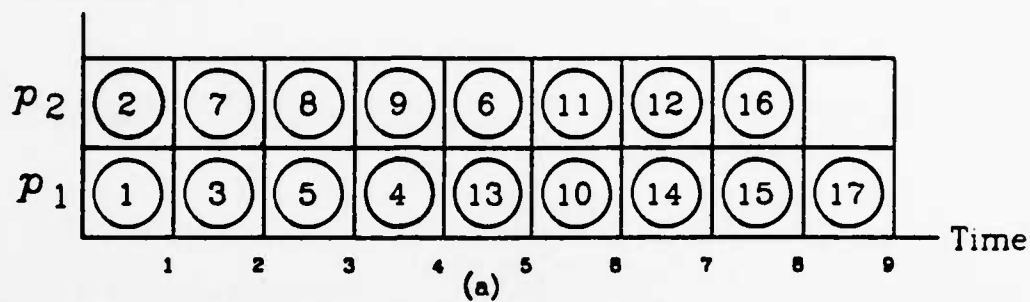
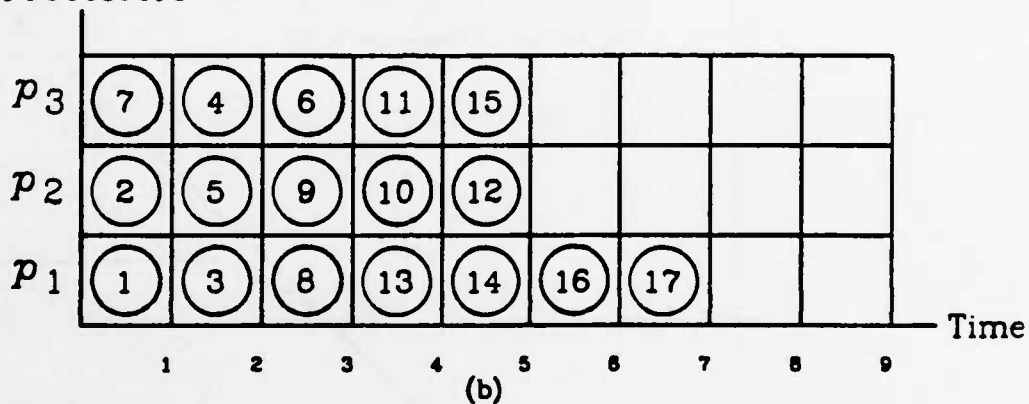


Figure 3.1 Example of a Directed Graph with Level Labeled for Each Node

Processors



Processors



Processors

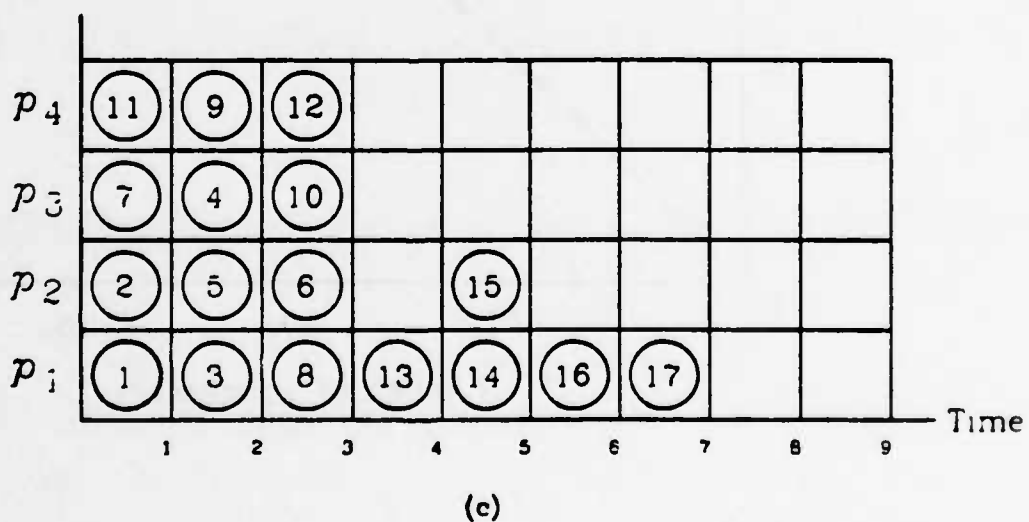


Figure 3.2

Schedules Obtained by Hu's Level Scheduling for the Directed graph in Figure 3.1 with (a) Two Processors (b) Three Processors (c) Four Processors

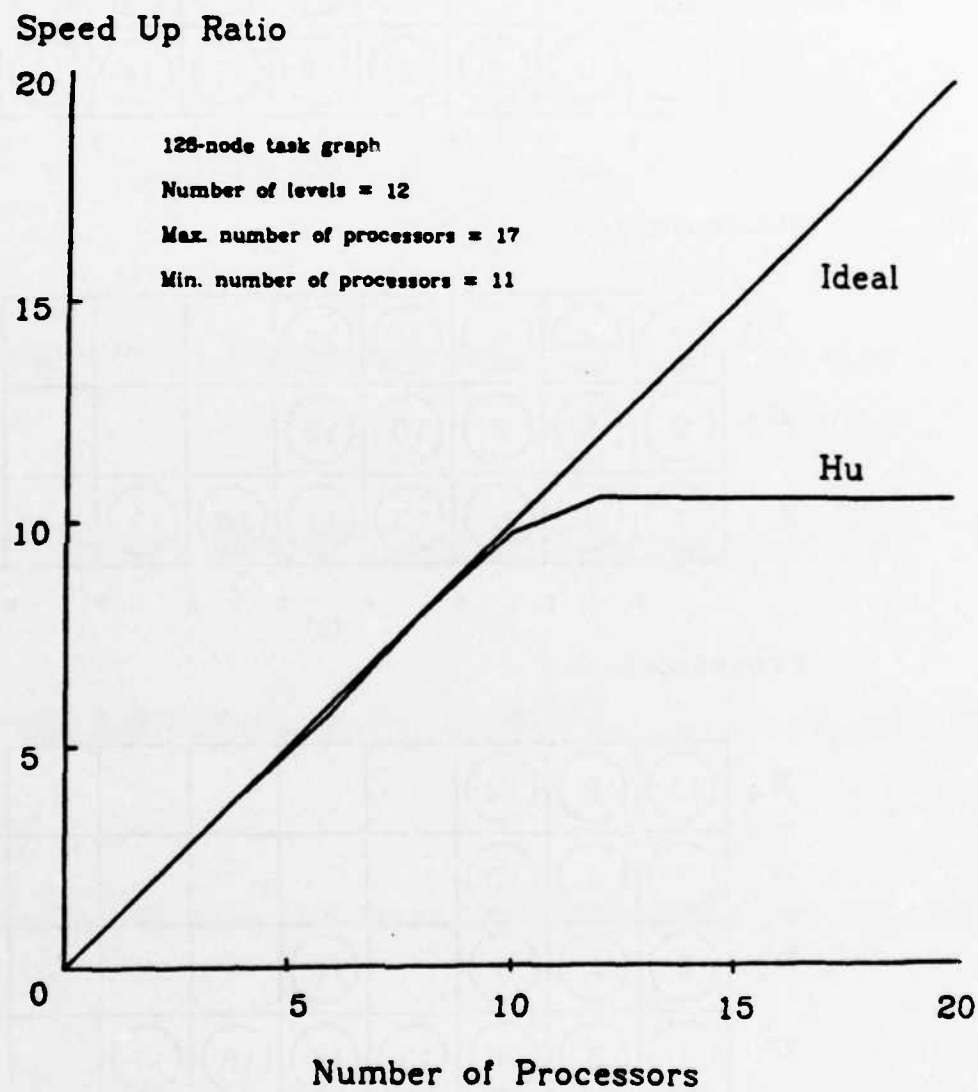


Figure 3.3 Speed Up Ratio Achieved Using Hu's Level Scheduling for the 128-Node Task Graph



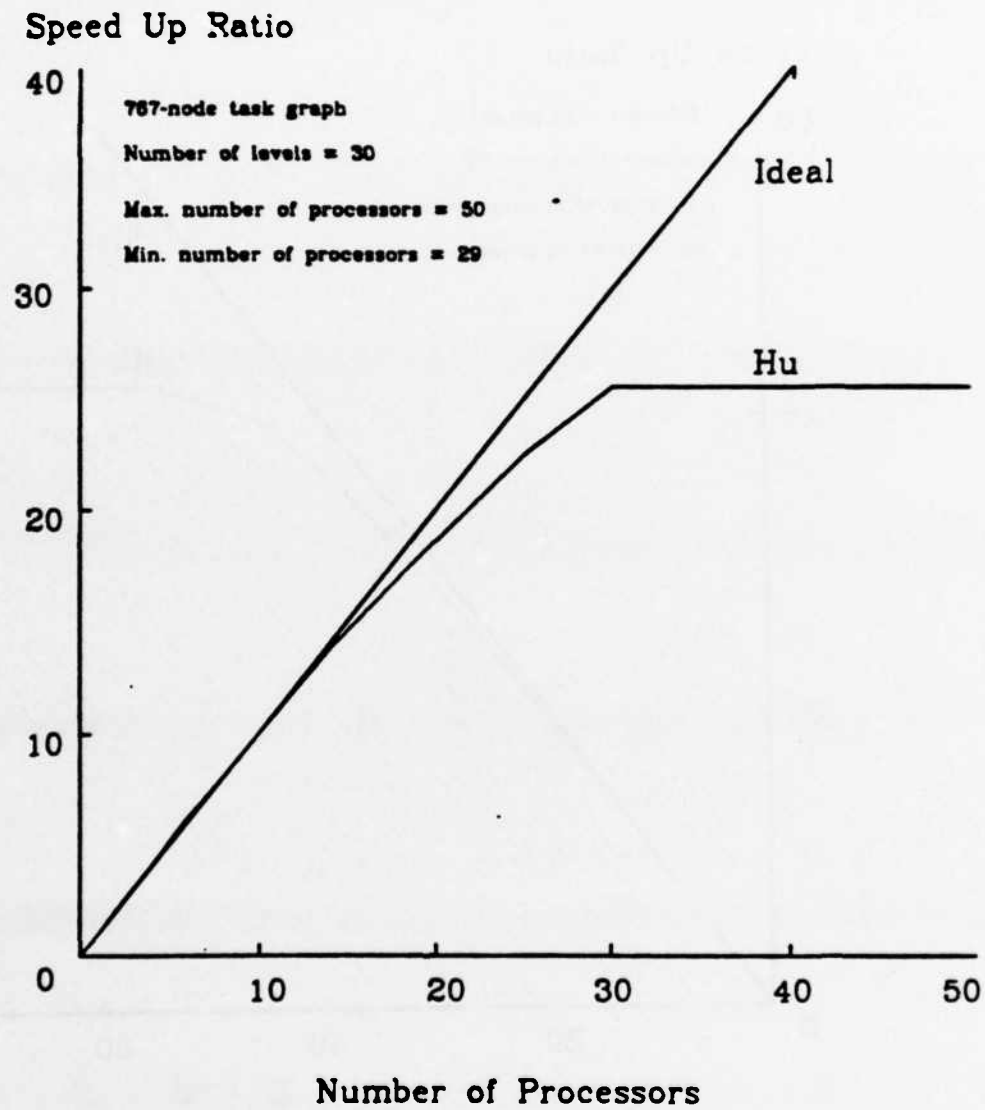


Figure 3.4(a) Speed Up Ratio Achieved Using Hu's Level Scheduling for the 767-Node Task Graph

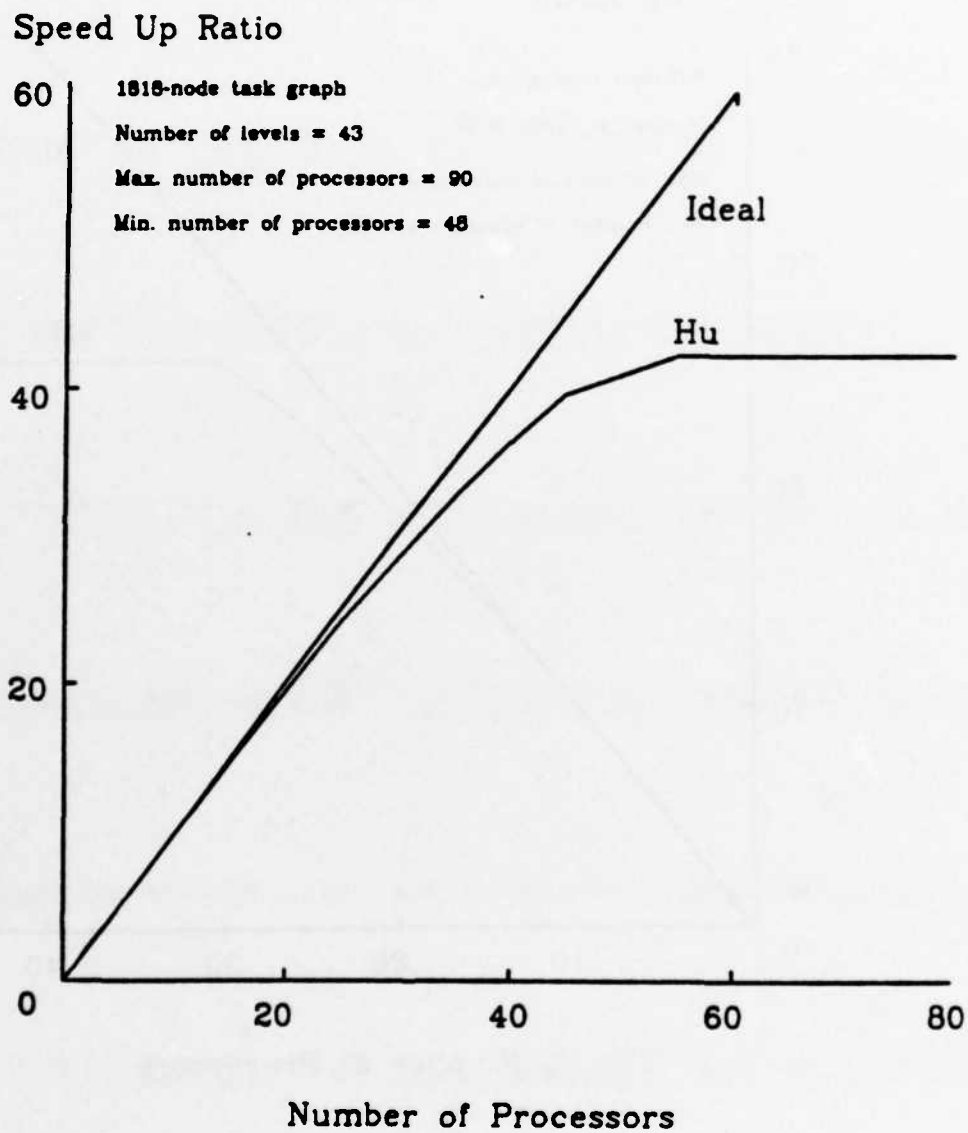


Figure 3.4(b) Speed Up Ratio Achieved Using Hu's Level Scheduling for the 1818-Node Task Graph

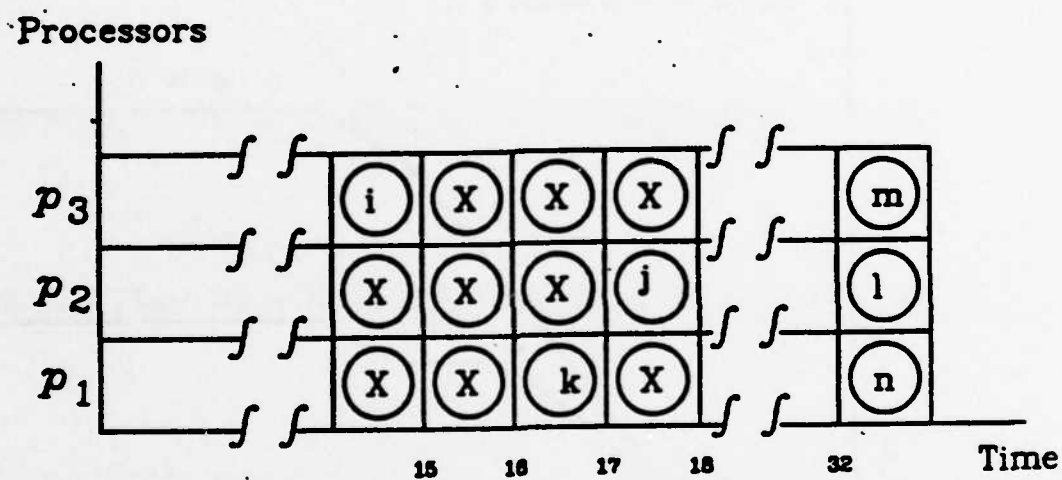


Figure 3.5

Segment of a Three-Processor Schedule at Timestep 32 Used to Illustrate the Determination of Completion Time of a Task Graph

## Speed Up Ratio

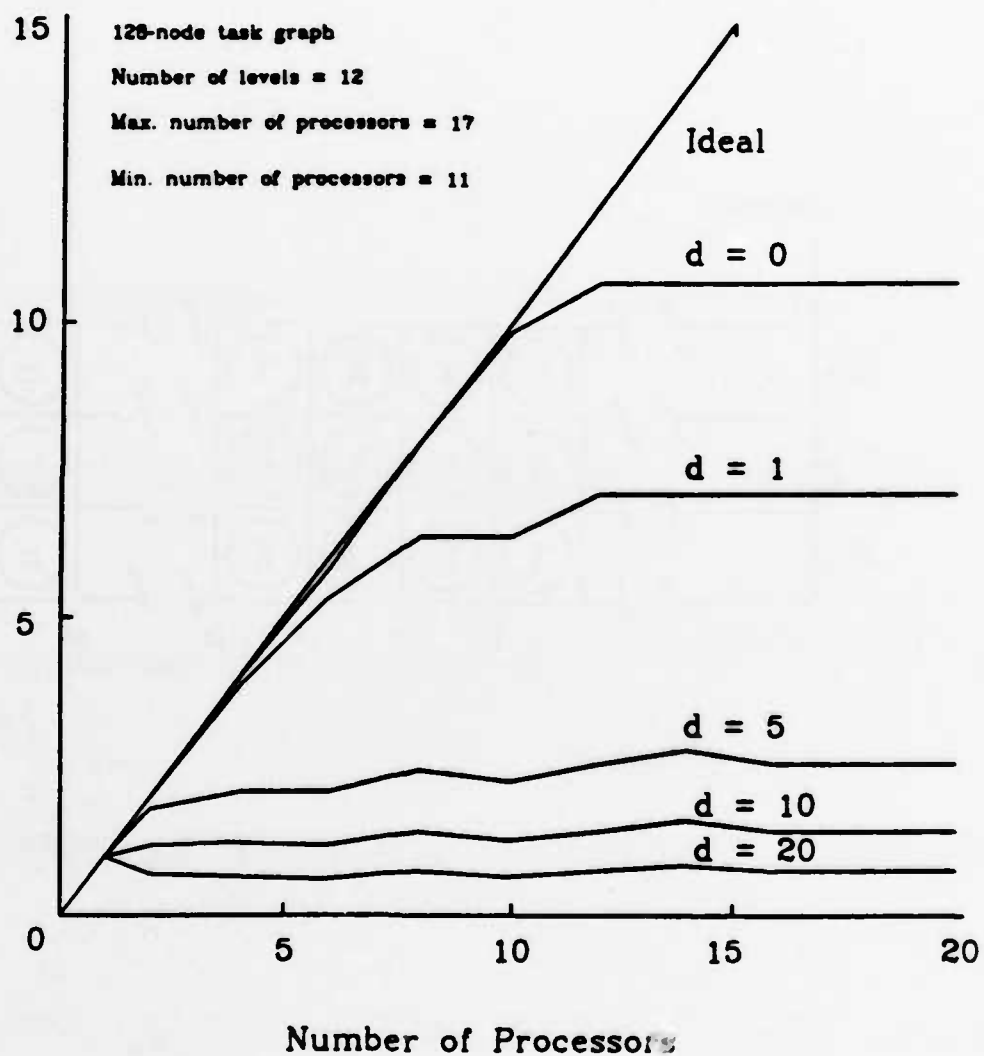


Figure 3.6(a) Speed Up Ratio Achieved Using Hu's Level Scheduling with Different Constant Delays for the 128-Node Task Graph

## Speed Up Ratio

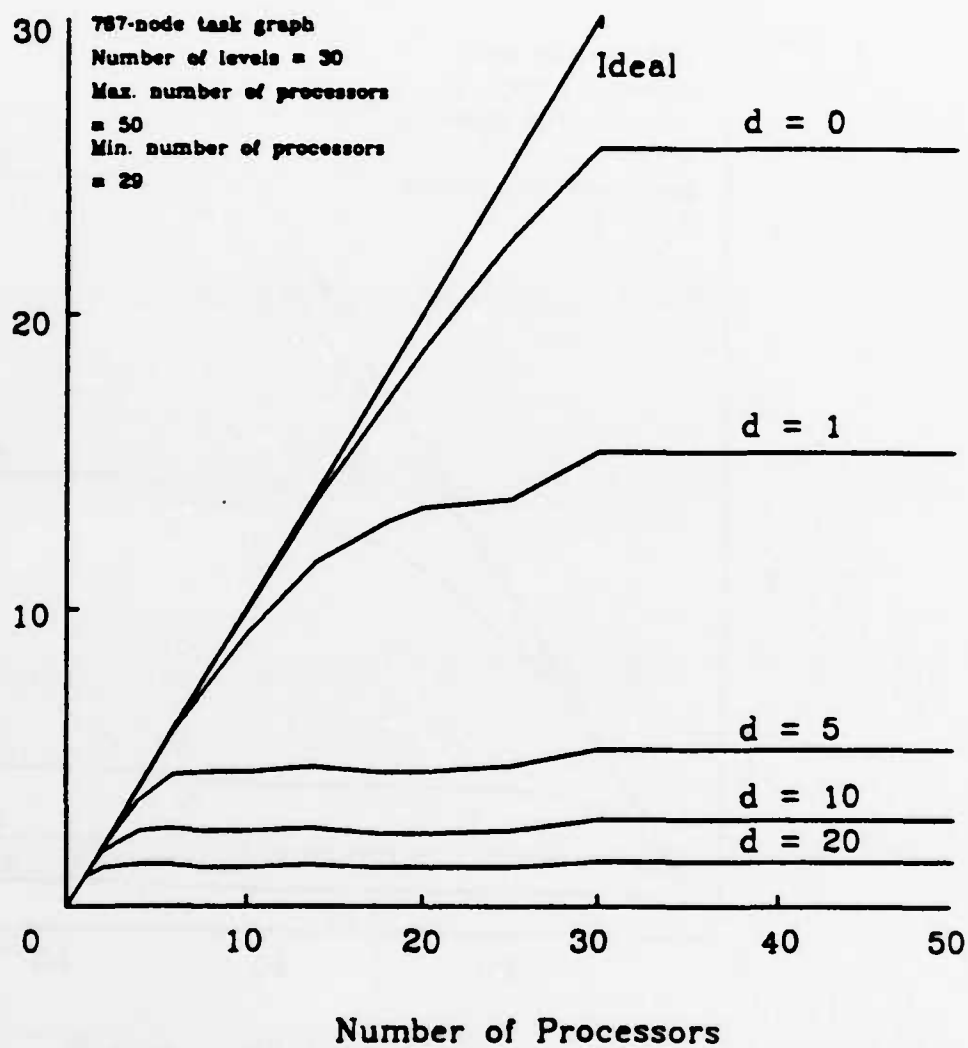


Figure 3.8(b) Speed Up Ratio Achieved Using Hu's Level Scheduling with Different Constant Delays for the 767-Node Task Graph

## Speed Up Ratio

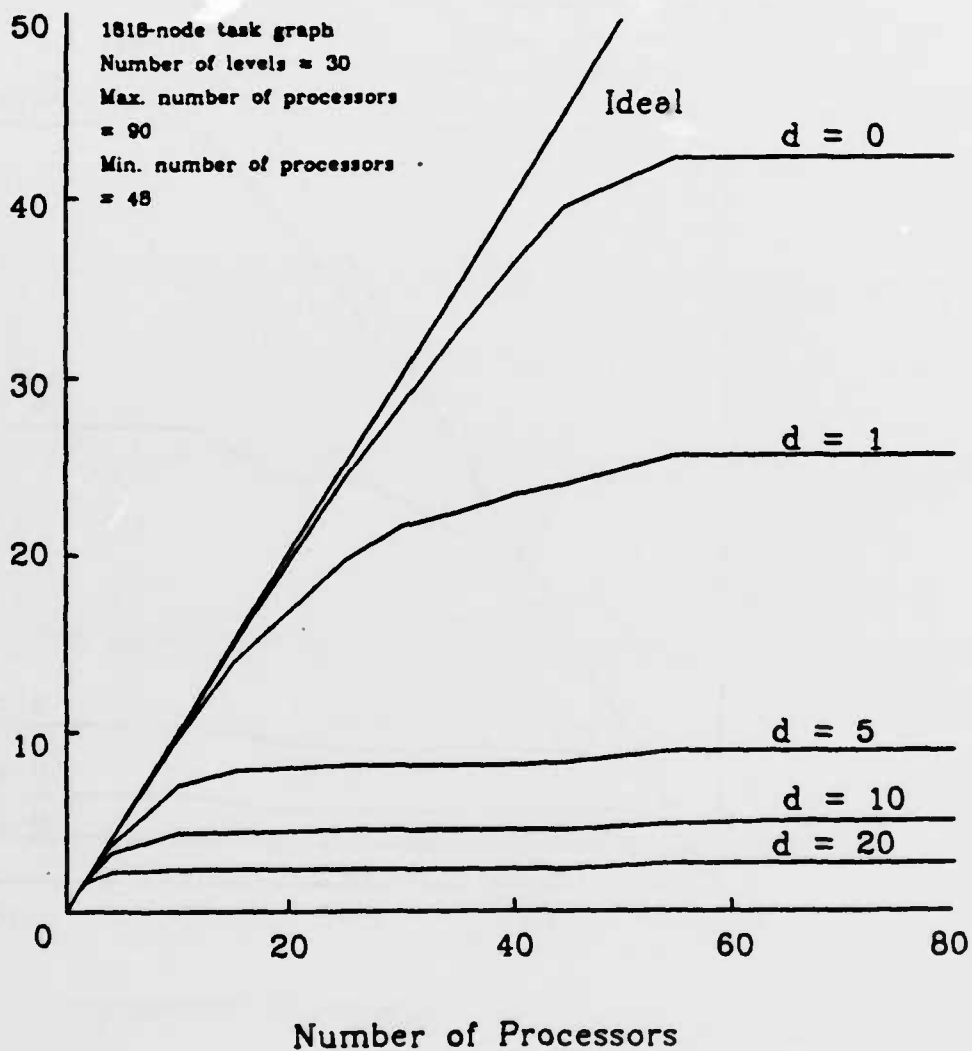


Figure 3.8(c) Speed Up Ratio Achieved Using Hu's Level Scheduling with Different Constant Delays for the 1818-Node Task Graph

## CHAPTER 4

### HEURISTIC LOCAL SCHEDULING TECHNIQUES

#### 4.1. Introduction

We have seen from the previous chapter that large communication delay compared to the node- execution time degrades the speedup performance considerably. In order to increase the efficiency of the LU decomposition algorithm in a multiprocessor system, scheduling algorithms taking into account the communication delay should be employed. In this and in the following chapters, scheduling algorithms are developed to minimize the completion time of a given LU task graph assuming constant delay on each edge of the task graph. The scheduling techniques are divided into two classes. One class is the *local* approach which is discussed in this chapter. The other class is the *global* approach which will be discussed in the next chapter.

The word *local* describes the scheduling techniques which minimize the completion time of the task graph *at each timestep* by performing combinatorial matchings of nodes ready for assignment to processors at that timestep. The word *global* refers to the scheduling technique which minimizes the completion time by considering all the nodes as a whole in the task graph. In this chapter, four heuristic local scheduling techniques are discussed. The performance of these heuristics is measured as the percentage of improvement over Hu's scheduling technique in the presence of communication delay. The precise definition of performance improvement will be defined as we discuss the simulation results for these local heuristics at the end of the chapter. Before we go into the details about each of the local heuristic scheduling techniques, let us get a glimpse of the complexity and the difficulty of the problem involved.



#### 4.2. Complexity and Difficulty of the Problem

Any sequencing algorithm whose complexity is bounded by a polynomial in the size of the problem is called polynomial-time algorithm. However there exists a class of problems called *NP-complete* problems which are equivalent in the following sense. If you can find a polynomial-time algorithm to solve one of these problems, one can solve essentially all the other problems in that class. Numerous efforts have been spent over the years unsuccessfully in the search for less than exponential time solution to these problems. There is strong evidence that these *NP-complete* problems are indeed inherently intractable. In the domain of scheduling theory, almost all sequencing problems stated in their complete generality fall into this category. In particular, it was shown that in each of the following cases, determining an optimal nonpreemptive schedule is *NP-complete*. The proof is given in [24].

- (1) The execution time of tasks ( nodes ) is arbitrary, there are two or more processors available.
- (2) The execution time of tasks ( nodes ) is one time unit, the precedence relation among tasks ( nodes ) is arbitrary, there are two or more processors available.

The scheduling of nodes of a LU task graph is similar to case (2) above. The presence of delay between communicating processors complicates the determination of the optimal schedule further. With the understanding of the complexity and the difficulty of the problem we are facing, a sensible approach is rather than finding the optimal schedule, try to develop some heuristic algorithms with feasible running time and which produce reasonable schedules, in the sense that these schedules will be better than Hu's level scheduling algorithm in the presence of communication delay.

In the following sections, four heuristic algorithms are described. For the purpose of comparison, they are named heuristic algorithms D, E, F and EF. Each of these algorithms will be discussed in detail. Some definitions and terminology pertaining to these heuristics are given in the following sections.

#### 4.3. Definitions and Terminologies

A *set of ready nodes* is a collection of nodes of a given graph which are ready for assignment to available processors. A node is in this set if its predecessors have been executed. In other words, its predecessors have been assigned to processors in the previous timesteps.

The *completion time* of a node is the time at which the processor to which this node is assigned finishes the execution of this node.

The *elapsed time* of a processor is the time at which it completes the execution of all the nodes assigned to it. The elapsed time includes the transmission delay as a result of having the predecessors of the node scheduled to different processors.

The *partial completion time* of a task graph at each timestep is the time required by all processors to complete the execution of all the nodes assigned to the processors so far. This partial completion time again includes the transmission delay and it is equal to the maximum of all the elapsed times of the processors. If at the last timestep, all the nodes in the task graph have been scheduled, then the partial completion time at this timestep will be the completion time of the task graph.

The following simple example schedule at timestep  $t_4$  will illustrate the above definitions. Suppose at timestep  $t_4$ ,  $\{i_1, i_2, i_3\}$  is a set of ready nodes and we have three processors  $p_1, p_2, p_3$ . The predecessors of node  $i_1$  are nodes  $j_1$  and  $k_1$ . The predecessors of node  $i_2$  is node  $j_2$  and the predecessor of node  $i_3$  is node

$j_3$ . ( See Figure 4.1(a) ) Assume that these predecessors are assigned as shown in Figure 4.1(b). Suppose the delay is  $d$  units and each node has an execution of one unit. Suppose further that an assignment is made such that node  $i_1$  is assigned to processor  $p_1$ , node  $i_2$  is assigned to processor  $p_2$  and node  $i_3$  is assigned to processor  $p_3$ . Let  $et(p_i)$  denote the elapsed time of processor  $p_i$ , then  $et(p_1) = t_i + d + 1$  ( delay in transmitting the result from node  $k_1$  in processor  $p_2$  to node  $i_1$  in processor  $p_1$  plus the node execution time which is one ),  $et(p_2) = t_i + d + 1$  ( delay in transmitting the result from node  $j_2$  in processor  $p_3$  to node  $i_2$  in processor  $p_2$  plus the node execution time which is one ), and  $et(p_3) = t_i + 1$  ( no delay because predecessor of node  $i_3$ , which is  $j_3$  is assigned to the same processor as node  $i_3$  ). The partial completion time at timestep  $t_i$  is the maximum of  $\{ et(p_1), et(p_2), et(p_3) \}$ , which is  $t_i + d + 1$ . Let  $ct(i)$  denote the completion time of node  $i$ , then  $ct(i_1) = et(p_1)$ ,  $ct(i_2) = et(p_2)$ ,  $ct(i_3) = et(p_3)$ .

In the later discussion of local heuristic algorithms E, F and EF, the assignment of nodes to processors is viewed as a classical matching problem in a special graph[25]. Hence some definitions in this respect are necessary and simple examples are used to illustrate these definitions.

A *bipartite graph* is a graph whose nodes can be partitioned into two sets  $S$  and  $T$ , so that each edge has one end in  $S$  and the other end in  $T$ . The edge between node  $i$  and node  $j$  is denoted by  $edge(i, j)$ . An example of a bipartite graph is shown in Figure 4.2(a).

Let  $G = (S, T, A)$  be an undirected bipartite graph. A subset  $X$  of  $A$  is said to be a *matching* if no two edges in  $X$  are incident to the same node. For example, in Figure 4.2(b), the wavy edges are in a matching. That is  $X = \{ edge(1,4), edge(2,6) \}$ .

With respect to a given matching  $X$ , a node  $i$  is said to be *covered* if there is an

edge in  $X$  incident to  $i$ . If a node is not covered, it is said to be *exposed*. In Figure 4.2(b), nodes (1), (2), (4) and (6) are covered while nodes (3) and (5) are exposed.

For a given matching  $X$ , an *alternating path* is a path of edges which are alternating in  $X$  and not in  $X$ . An *augmenting path* is an alternating path between two exposed nodes. The size of a matching is measured by the number of edges in the matching. By discovering an augmenting path, the size of the matching can be increased by one. For example, in the graph of Figure 4.2(b), a path = { (3), (4), (1), (5) } is an augmenting path with nodes 3 and 5 being the exposed nodes. This augmenting path is also shown in Figure 4.3(a). The following procedure explains why it is an augmenting path. The matching  $X$  of the bipartite graph shown in Figure 4.2(b) contains  $edge(1,4)$  and  $edge(2,6)$ . Hence  $X$  has a size equal to two. By excluding  $edge(1,4)$  in  $X$  and putting  $edge(3,4)$  and  $edge(1,5)$  into  $X$  as shown in Figure 4.3, the new matching now contains  $edge(3,4)$ ,  $edge(1,5)$  and  $edge(2,6)$ . This new matching is shown in Figure 4.4. The size of new matching is now equal to three. Hence by finding the augmenting path of { (3), (4), (1), (5) }, we are able to increase the size of the matching by one.

As mentioned earlier in this chapter, three of the local heuristic schemes use combinatorial optimization techniques to obtain an assignment of nodes to processors. Two matching algorithms are employed to obtain such an assignment. They are the min\_max matching algorithm and weighted matching algorithm. The detail description of these two matching algorithms can be found in standard textbook such as [25]. For completeness, they are stated below.

#### *Min\_Max Matching Problem*

Given an edge weighted bipartite graph, find a matching containing a maximum number of edges for which the maximum weights of the edges in the matching is minimum. The min\_max matching algorithm uses the *Augmenting Path Theorem*.

which states that a matching  $X$  contains a maximum number of edges if and only if it admits no augmenting paths. The algorithm for finding the matching is called *Threshold Method* for min\_max matching problem and it is described in[25]. The other matching algorithm is called *the Weighted Matching Problem* and is stated below:

*Weighted Matching Problem*

Given an edge-weighted bipartite graph, find a matching for which the sum of the weights of the edges is maximum. The algorithm for obtaining the matching is also described in[25].

When these two matching algorithms are applied to these local heuristic techniques, the nodes-to-processors scheduling environment with the consideration of communication delay should be mapped into a weighted bipartite graph. The detail of this correspondence will be discussed as we describe the heuristics. With these definitions and terminologies in mind, we are ready to describe the four local heuristic scheduling techniques in the following sections.

#### 4.4. Heuristic Algorithm D

This is a very intuitive approach which does not involve any of the combinatorial techniques. Our main objective is to minimize the partial completion time at each timestep. Given a set of ready nodes, a node with the highest level is chosen from the set for assignment to processors. Suppose we are given  $p$  processors, the selected node is assigned to these processors one at a time and the elapsed time of each processor is found by taking into account the delay when the predecessors of the chosen node are located in different processors. For this particular node, we have  $p$  such elapsed times. We can view these elapsed times as a vector  $ET = [et_1, \dots, et_p, \dots, et_p]^T$  where  $et_i$  is the elapsed time of processor  $i$  when this chosen node is assigned to it. Among these  $p$  entries in the vector, choose the minimum, say  $et_p$  is the minimum, and assign this node to

processor  $\beta$ . If there are more than one minimum, pick one arbitrarily. Then another node with the next highest level is chosen from the set of ready nodes. At this time, we are left with  $(p - 1)$  processors to which we can try to assign this node. So the vector of elapsed time  $ET = [et_1, \dots, et_{\beta-1}, et_{\beta+1}, \dots, et_p]^T$  has only  $(p - 1)$  entries. Assign this node to the processors with the minimum elapsed time and so on. Continue this process until we have finished assigning all the nodes in the ready set or there are no more processors available at this timestep. Then find another set of ready nodes. Repeat the above assignment procedure until all the nodes in the task graph are scheduled. The high level description of the program is given below:

```
timestep = 0 ;
while ( there are unassigned nodes )
{
  (1) Find the set of ready nodes at this timestep.
  (2) Pick a node with the highest level from this set of ready nodes.
  (3) Assign the chosen node to each of the available processors and find all
      the elapsed time of these processors.
  (4) Among all the elapsed times, assign the chosen node to the processor
      which gives the minimum elapsed time. If there are more than one pro-
      cessor having the same minimum. Choose one arbitrarily.
  (5) Reduce the number of available processors by one. Remove the node
      from the set of ready nodes.
  (6) If ( there are still processors available and the set of ready nodes is
      nonempty )      Go back to (2).
      Else      Increment the timestep.
} /* end of while */
```

The advantage of this heuristic algorithm is simple and the running time is proportional to the number of nodes in the task graph. A moment's thought reveals the fact that the above assignment will not always give the shortest partial completion time at each timestep. The obvious reason is that we do not consider all the nodes in the ready-node set at the same time. Let us consider a simple and hypothetical case. Suppose we have two processors available and at timestep equal to 3, there are only two nodes, nodes (3) and (4) in the set of ready nodes and they are of the same level. Hence there are only two possible schedules as shown in Figure 4.5(a) and Figure 4.5(b). The elapsed time of the processors are as shown. In schedule (I), the elapsed time of  $p_1$  is 5 and the elapsed time of  $p_2$  is 11. In schedule (II), the elapsed time of  $p_1$  is 4 and the elapsed time of  $p_2$  is 8. If node (3) is selected first, we have schedule (I) and on the other hand if node (4) is selected first, we have schedule (II). Given the two possible assignments (I) and (II), it is clear that schedule (II) is better than schedule (I) in the sense that schedule (II) has a shorter partial completion time ( equal to 8 in this example ) than schedule (I) ( equal to 11 in this example ). Hence the order in which nodes of the same level from the set of ready nodes is chosen may affect the final completion time of a task graph. In order to minimize this effect and to get the shortest partial completion time at each timestep, all possible nodes-to-processors combinations have to be considered and among all these possible assignments, we choose the one which gives the shortest partial completion time. However the time it takes to obtain all these possible combinations is proportional to the factorial of the number of processors available as well as the number of nodes ready to be scheduled at that timestep. To be specific, let  $p$  be the number of processors available and let  $n$  be the number of nodes in the set of ready nodes at a certain timestep. There are  $n(n-1)(n-2)\dots(n-p+1) = \frac{n!}{(n-p)!}$  possible schedules to be considered.



It is obvious that this exhaustive method is not a good approach. Therefore some combinatorial matching algorithms should be employed to minimize the time required to search for optimal nodes-to-processors assignments. The *min\_max* matching is well suited for the above purpose. It will find a matching ( between nodes and processors ) to obtain such an assignment.

However, there is still room for us to optimize in order to get a better schedule. We will illustrate this point again based on the following example. There are cases in which more than one schedule is obtained with the same minimum partial completion time at a certain timestep. The question is, among these minimum partial completion time schedules, is there one better than the other ? Let us look at the following simple example in which there are two nodes in the set of ready nodes at timestep  $t_4$  equal to 4, and there are only two available processors. Suppose we have the following two possible schedules as shown in Figure 4.6. Both schedules have the same partial completion time ( at 13 time units ). The *min\_max* matching algorithm applied at this timestep will give either schedule (I) or schedule (II). However it may be beneficial to have schedule (I) because the sum of the elapsed times of the two processors  $p_1$  and  $p_2$  is less than that of the sum of the elapsed times of the two processors in schedule (II). In this case, it may be possible to assigned more nodes in schedule (I) to processor  $p_1$  after the execution of node (5) than to processor  $p_2$  in schedule (II). In other words, since node 5 has a shorter completion time ( at 5 time units ) in  $p_1$  of schedule (I) than the completion time ( at 9 time units ) in  $p_2$  of schedule (II), more nodes can be assigned to  $p_1$  in schedule (I) after timestep equal to 5. Hence we would like to have a more *compact* schedule at each timestep so that more nodes can be assigned at the later timestep and thus reduce the completion time of the task graph.

These two matching algorithms are incorporated in the next three local

heuristic scheduling algorithms. The detailed mapping of the nodes-to-processors scheduling environment into a bipartite graph so that these matching algorithms can be applied will be described as we discuss the heuristics in the next few sections.

#### 4.5. Application of Min\_Max and Weighted Matchings to Heuristic Algorithms

In the next three heuristic algorithms E, F and EF, it is necessary to obtain a weighted bipartite graph at each timestep representing the scheduling environment. Let us describe how it is done. Suppose there are  $p$  processors available and at a certain timestep, there are  $n$  nodes in the set of ready nodes. Pick  $\min\{p, n\}$  nodes from this set and assign each node to one of the  $p$  processors, find the elapsed time of each processor taking into account the communication delay due to the fact that the predecessors of that node are assigned to different processors. After this is done, we have represented this two-dimensional data in a matrix of size  $p$  by  $\min\{p, n\}$ . The row index in the matrix corresponds to processor number and the column index corresponds to the node number. The value of the  $(i, j)^{th}$  element, denoted by  $et_{ij}$  in the matrix will be the elapsed time of processor  $i$  when node  $j$  is assigned to it. With the previous notation of a bipartite graph  $G = (S, T, A)$ , we can consider the processors  $p_1, p_2, \dots, p_p$  as the nodes in the set  $S$ , the nodes  $i_1, i_2, \dots, i_n$  in the set of ready nodes as the nodes in  $T$ , and the elapsed times will be the weights on the edges of the bipartite graph. A simple bipartite graph representing the scheduling environment is shown in Figure 4.7. There are two processors  $p_1, p_2$  available and there are three nodes  $i_1, i_2, i_3$  ready for assignment at a certain timestep. The matrix representation,  $ET = [et_{ij}]$  of the weighted bipartite graph of Figure 4.7 is shown below :

$$ET = \begin{bmatrix} et_{11} & et_{12} & et_{13} \\ et_{21} & et_{22} & et_{23} \end{bmatrix}$$

where  $et_{ij}$  is the elapsed time of processor  $i$  when node  $j$  is assigned to it.

Although these three heuristic algorithms are based on the combination of the min\_max matching and weighted matching algorithms applied at each timestep, it is not a straight application of min\_max matching followed by weighted matching. A slight modification of the weights of the bipartite graph is needed after a matching from the min\_max algorithm is obtained. The reason is that the weighted matching algorithm will find a nodes-to-processors assignment with the sum of the elapsed times of the processors being maximum. However, what we would like to have is an assignment such that the sum of the elapsed times of the processors is minimum. Hence the entries which are the elapsed times in the matrix ( bipartite graph ) obtained from the min\_max matching algorithm are modified as follow:

$$\begin{aligned} \max\_ent &= \max_{i,j} (et_{ij}) \times \text{Min}(p, n) \\ et_{ij} &= \max\_ent \times \text{Min}(p, n) - et_{ij} \end{aligned} \quad (4.1)$$

#### 4.5.1. Simple Illustrative Examples

The following two examples will illustrate how min\_max and weighted matchings can be applied to obtain the heuristic algorithms described in the next three sections. Consider the following scheduling environment at time equal to  $t_1$ . Suppose there are two processors  $p_1, p_2$  available and there are two nodes  $i_1, i_2$  ready for assignment. Hence there are only two possible assignments. One assignment will have node  $i_1$  assigned to  $p_1$  and node  $i_2$  assigned to  $p_2$ . The other assignment will have node  $i_1$  assigned to  $p_2$  and node  $i_2$  assigned to  $p_1$ . Further suppose the elapsed times of the processors are shown in the following matrix,  $ET = [ et_{ij} ]$ :

$$ET = \begin{bmatrix} 1 & 3 \\ 2 & 5 \end{bmatrix}$$

The above matrix,  $ET$  is a representation of a weighted bipartite graph shown in

Figure 4.8(a). When min\_max matching algorithm is applied to this weighted bipartite graph ( the detail procedure of the algorithm can be found in [25] ), the matching,  $X$  containing  $edge(p_1, i_2)$  and  $edge(p_2, i_1)$  shown in Figure 4.8(b) is obtained. This represents the assignment of node  $i_1$  to processor  $p_2$  and node  $i_2$  to processor  $p_1$ . This assignment has a shorter partial completion time of 3 units ( equal to  $\text{Max} \{ et_{12} = 3, et_{21} = 2 \}$  ) as compared to the other assignment which has a partial completion time of 5 units ( equal to  $\text{Max} \{ et_{11} = 1, et_{22} = 5 \}$  ). Hence the min\_max matching algorithm will always give an assignment which has the shortest partial completion time at a timestep.

However, it is sometimes possible to have more than one assignment with the same shortest partial completion time. Consider the following scheduling environment with two processors  $p_1, p_2$  and two nodes 5, 8 ready for assignment at timestep equal to 4. Further suppose that the elapsed times of the processors are shown in the following matrix,  $ET = [ et_{ij} ]$ :

$$ET = \begin{bmatrix} 5 & 13 \\ 9 & 13 \end{bmatrix}$$

The corresponding weighted bipartite graph is shown in Figure 4.9(a). The two possible schedules are shown in Figure 4.6. The min\_max matching will result either a matching shown in Figure 4.9(b) which corresponds to the schedule (I) of Figure 4.6, or a matching shown in Figure 4.9(c) which corresponds to the schedule (II) of Figure 4.6. Note that these two possible assignments have partial completion time of 13 units. As discussed in detail at the end of section 4.4, a *compact* schedule ( schedule whose sum of the elapsed times of the processors is smallest ) is preferred because more nodes can be assigned at the later timestep ( more detailed explanation is found at the end of section 4.4 ). Hence schedule (I) of Figure 4.6 is a better choice than schedule (II) in the same figure. In schedule (I), the sum of the elapsed times of the two processors is 18 units while in schedule (II), the sum of the elapsed times of the two processors is 22

units.

The problem is how to obtain an assignment such that the partial completion time at a certain timestep is minimum and in case there are more than one assignment with the same partial completion time, choose the assignment whose sum of the elapsed times of the processors is minimum. To obtain the minimum partial completion time, we apply the min\_max matching as explained in the last paragraph. To obtain the smallest sum of elapsed times of processors, weighted matching is employed. Since weighted matching will find a matching such that the sum of the weights ( which are the elapsed times in this application ) on the edges is maximum, and the scheduling assignment requires the sum of the weights to be minimum, the weights on the bipartite graph should be modified. Essentially, we have to change the smaller weights to larger weights and larger weights to smaller weights on the bipartite graph. This can be accomplished by modifying the entries of the matrix  $ET$  according to equation (4.1). Hence in this example, the matrix of the modified graph will be as follow :

$$ET = \begin{bmatrix} 21 & 13 \\ 17 & 13 \end{bmatrix}$$

Applying the weighted matching to the modified bipartite graph, we will obtain the matching shown in Figure 4.9(b) which corresponds to the schedule (I) of Figure 4.6.

The heuristic algorithms E, F and EF are based on the application of these two combinatorial algorithms. Nodes from the set of ready nodes are chosen at each timestep and min\_max and weighted matchings are employed. The difference between these heuristics is the criterion we are using to select the candidate nodes for assignment to processors.

#### 4.6. Heuristic Algorithm E

In this heuristic, the candidate nodes are the nodes with the highest level taken from the set of ready nodes. The high level description of the program is given below:

```

timestep = 0 ;      p processors are available
while ( there are unassigned nodes )
{
  (1) Obtain a set of ready nodes.

  (2) Count the number of ready nodes at this timestep, say it is equal to m.

  (3) If ( m > p ) choose p nodes from the set of ready nodes with highest
      level
      Else      choose all m nodes from the set.

  (3) Form the matrix (p by min(p,m) ) whose entries are the elapsed times
      of the processors.

  (4) Find the maximum value of the entry in the matrix, say it is equal to
      max_ent.

  (5) Find the minimum value of the entry in the matrix, say it is equal to
      min_ent.

  (6) threshold = min_ent.

  (7) for ( ; ; ) /* loop for min_max */
      { find all edges whose weights = threshold ; /* we have a bipartite
        graph */
        find augmenting paths in the graph ;
        update the matching ;
        find the number of edges in the matching ; say it is equal to num_edge ;
        if ( num_edge = min ( p, m ) ) break ; /* done with min_max */
        else      threshold ++ ;
      }
}

```

```
    } /* end of loop for min_max */
```

(8) For this most current matrix ( bipartite graph ), update each entry as described in the last section.

(9) Apply weighted matching algorithm to this matrix to obtain the final assignment.

(10) Increment the timestep.

```
    } /* end of while loop */
```

In summary, in this heuristic algorithm E, only the nodes with the highest level are the candidates to be scheduled to processors. Hence it is not much different from the level scheduling technique. Also in the case where the number of nodes in the set of ready nodes at a timestep is larger than the number of processors available, we only choose the number of nodes equal to the number of processors. By excluding some nodes from the set of ready nodes, this may lead to an assignment which will not give the shortest partial completion time at this timestep. In the next section, we will discuss another heuristic algorithm F which is a modification of the heuristic E.

#### 4.7. Heuristic Algorithm F

The modification to the previous heuristic algorithm is based on the fact that we should consider all the nodes in the set of ready nodes no matter how many nodes are there in that set. Hence the concept of the level of a node is not incorporated in this heuristic algorithm. This heuristic scheduling algorithm considers all the nodes ready for assignment to all the available processors taking into account the communication delay between processors and then it finds an assignment which gives the smallest partial completion time at that timestep. If the size of the set of ready nodes is larger than the number of processors available, the candidate nodes are not necessarily restricted to the



nodes with the highest level. If the size of the set of ready nodes is smaller than the number of processors available at each timestep, then heuristic F is identical to heuristic E. The heuristic algorithm F is described below :

```
timestep = 0 ; p processors are available
while ( there are unassigned nodes )
{
  (1) Obtain a set of ready nodes.
  (2) Count the number of nodes in this set, say it is equal to  $m$ .
  (3) Form the matrix ( bipartite graph ) of  $p$  by  $m$ .
  (4) Apply min_max matching algorithm.
  (5) Apply weighted matching on the modified bipartite graph as discussed
      before to obtain the final assignment.
  (6) Increment the timestep.
} /* end of while */
```

Beside the motivation of considering all the nodes in the set of ready nodes so that all possible assignments are taken into account, the other reason is that we will get a more "compact" schedule in which processors will have less idle time. In this case, nodes with lower level may get scheduled ahead of those nodes with higher level. One might think that by delaying the execution of nodes with higher level will increase the final completion time of the task graph. This is true in the case where there is no communication delay or when the delay is small compared to the node execution time. In the situation where large delay exists between communicating processors, if we only pick nodes with high level as the candidate nodes, they may have large node completion times resulting a long idle times in the processors. It may be beneficial to schedule nodes with smaller node completion times first at this timestep to "fill" in those idle times

even though these nodes are of lower level.

In summary, this heuristic is solely based on the fact that shortest partial completion time of processors is the primary concern without regarding to the level of a node. It is aimed at handling the situation where large delay compared to the node execution time exists between communicating processors. In the next section, another heuristic algorithm EF will be discussed. It represents another possible approach between these two extremes in which the candidate nodes must have high level ( heuristic E ) and the candidate nodes should have short node completion times ( heuristic F ).

#### 4.8. Heuristic Algorithm EF

In the previous two sections, two different concepts are employed in two different local heuristic scheduling algorithms. One is based on the concept of levels of ready nodes, the other is based on the concept of the completion times of the ready nodes. In this heuristic EF, the candidate nodes for assignment to processors are based on the weighted sum of their levels and their node completion times. To be more specific, let  $\beta$  be a tuning parameter which has a value between zero and one, let  $lvl_i$  be the level of a given node  $i$  and let  $ct_{ij}$  be the completion time of node  $i$  when it is assigned to processor  $j$ . Then the entry  $s(i, j)$  in the matrix (i.e. weight on the bipartite graph) is assigned the following value :

$$s(i, j) = \beta \times \frac{lvl_i}{\max\_lvl} + (1 - \beta) \times ct_{ij} \quad 0 \leq \beta \leq 1$$

where  $\max\_lvl$  is the maximum level in the task graph. The program flow is identical to heuristic F except the entries in the matrix are computed differently.

In summary, this heuristic gives an alternative way of scheduling nodes in the presence of communication delay. It is expected on the average to perform

better than the previous two heuristics E and F. However we cannot say that it is always the case because of the heuristic nature of these algorithms. In concluding the description of all these four heuristic algorithms, an important question is the order of complexity of these heuristics. This will be discussed in the next section.

#### 4.9. Complexity of these Heuristic Algorithms

The first heuristic D described in this chapter is different from the remaining three heuristics in the sense that no combinatorial optimization technique is employed in heuristic D. Recall that in this heuristic, one node is scheduled from the set of ready nodes and it is assigned to a processor until all the nodes in the set are scheduled at this timestep. Then a new set of ready nodes is formed at the next timestep. The above process is repeated until all the nodes in the task graph are assigned. It is clear that the complexity of this simple heuristic algorithm is *linear* in the size of the task graph. That is  $O(n)$ , where  $n$  is the number of nodes in the graph.

The other three heuristics E, F and EF are based on the application of min\_max and weighted matching algorithms at each timestep. Let  $|S|$  denote the number of nodes in the set  $S$  of a bipartite graph. In [25], it is shown that if  $|S| = m$ , and  $|T| = n$ , the complexity of the threshold method for min\_max matching is  $O(m^2n^2)$  while for the weighted matching, the complexity is  $O(m^3n)$ . Suppose a given task graph has  $n$  nodes and there are  $p$  processors available, it requires  $O(\frac{n}{p})$  timesteps to schedule all  $n$  nodes in the task graph to the processors. At each timestep, the size of the set of ready nodes is bounded by  $n$ . Hence the order of complexity of performing min\_max and weighted matchings is  $O(n^2p^2 + n^3p)$ . Since it takes the order of  $(\frac{n}{p})$  timesteps to schedule all the nodes, the complexity will be  $O((\frac{n}{p}) \times (n^2p^2 + n^3p))$  which is

$$O(n^3p + n^4).$$

#### 4.10. Simulation Results and Discussion

Six sparse matrices extracted from six benchmark circuits in SPICE[1] are used to test the performance of the four heuristic algorithms. the performance is measured in terms of how much percentage of improvement in speedup ratio compared to Hu's level scheduling technique is achieved when we have constant delay between communicating processors.

$$\% \text{ of improvement} = \frac{\text{speedup ratio}(\text{Heuristic}) - \text{speedup ratio}(\text{Hu})}{\text{speedup ratio}(\text{Hu})} \times 100$$

These matrices are of different sizes ranging from a 12 by 12 matrix to a 32 by 32 matrix. The number of operations necessary for the LU decomposition ( number of nodes in the LU task graph ) ranges from 58 ( from the 12 by 12 matrix ) to 787 ( from the 32 by 32 matrix ). The number of processors and the delays are the two parameters used in testing out the performance of the heuristics. In addition, the tuning factor  $\beta$  described in heuristic EF is also a parameter in generating a family of curves. For the delay, we assume that each operation takes one unit execution time and the delays are integral multiple of the node execution time. Delays of 1, 5, 10, and 20 time units are used. Notice that these tests are performed with the understanding that the results are by no means representative. How well these heuristic algorithms perform depends on the structure of the task graph, the number of processors available and the value of delay. In other words, these algorithms might give better results for one graph than another. However, we expect that these heuristics, in general, will give better speedup ratio than Hu's level scheduling algorithm when there is communication delay between processors exchanging data.

The simulation results can be discussed in two categories. In one category, the performance of each heuristic with different delays is discussed. The perfor-

mance improvement for heuristic D applied to six graphs are shown in Figure 4.10 and Figure 4.11. The number of operations ( nodes ) for each graph is shown on each figure. The results for heuristic E, F and EF are shown from Figure 4.12 to Figure 4.13, from Figure 4.14 to Figure 4.15 and from Figure 4.16 to Figure 4.17 respectively. For heuristic D, E and F, in most of the cases, it is observed that as the delay increases, the percentage of improvement over Hu increases for a given number of processors. For heuristic EF, the same general trend is observed. However there are cases in which the delay increases and the performance decreases for a certain value of  $\beta$ . As discussed above, there is no definitive conclusion which says that for a given set of parameters such as  $\beta$ , delay and the number of processors, we will achieve a certain speedup ratio. But in nearly all of the simulation results, these heuristic algorithms do indeed tend to give better schedules than Hu's level scheduling technique. On the average, the heuristic D gives 5% to 15% of improvement, heuristic E gives 10% to 25%, heuristic F gives 20% to 35% and there are cases it gives 40% to 50% of improvement. For heuristic EF, it gives on the average 50% to 80% for some  $\beta$ . Sometimes for a certain value of  $\beta$ , we achieve over 100% of improvement. A final remark with respect to this category is that all these heuristics do not give much improvement especially heuristic F for *small* delay. In some cases, F gives schedules which have longer final completion time than Hu's. This may lead to the conclusion that probably Hu's level scheduling algorithm is one of the appropriate algorithms to use when there is small delay or when there is no delay at all.

In the other category, these four heuristic are compared with each other for a given delay. For heuristic EF, only three values of  $\beta$  ( 0.1, 0.5, 0.9 ) are chosen to compare with the other three heuristics. The performance comparison of these four heuristics for the graph with 58 nodes with different delays

are shown in Figure 4.18 and Figure 4.19. The results for the graphs with 128 nodes, 295 nodes, 844 nodes, 725 nodes and 767 nodes for different delays are shown from Figure 4.20 to Figure 4.21, from Figure 4.22 to Figure 4.23, from Figure 4.24 to Figure 4.25, from Figure 4.26 to Figure 4.27 and from Figure 4.28 to Figure 4.29 respectively. In general, F performs better than E which in turn performs better than D for large delay. As discussed in the last paragraph, in the case of small delay, F performs worse than E and D. This shows that for small delay, the level of a node should be used as a criterion for selecting candidate nodes while for large delay, the node completion time is the primary consideration in choosing the nodes. The heuristic EF is developed based on the above observation that it may be advantageous to consider the weighted sum of the level of a node and its node completion time. In fact it is shown in the results that there are some values of the tuning parameter  $\beta$  which give quite impressive speedup performance.

#### 4.11. Conclusion

In this chapter, four local heuristic scheduling algorithms are discussed. These heuristics are developed to assign nodes in a LU task graph to processors when there is a fixed communication delay between processors exchanging data. The heuristic D assigns nodes one at a time from the set of ready nodes at each timestep. The other three heuristics perform combinatorial matching between nodes in the node ready set and the processors to obtain an assignment at each timestep. In most of the cases, promising results in which performance is measured as the percentage of improvement in speedup ratio compared to Hu's level scheduling technique are obtained. Note that we are trying to achieve the shortest completion time of the task graph heuristically by minimizing the partial completion times of the processors at *each timestep*. In the next chapter, another technique is attempted which tries to minimize the completion time by

considering one aspect of the structure of a given task graph, namely the *critical path* of that graph. By taking into consideration *all* the nodes in the task graph, it is expected that this *global* technique gives shorter completion time than the techniques described in this chapter.



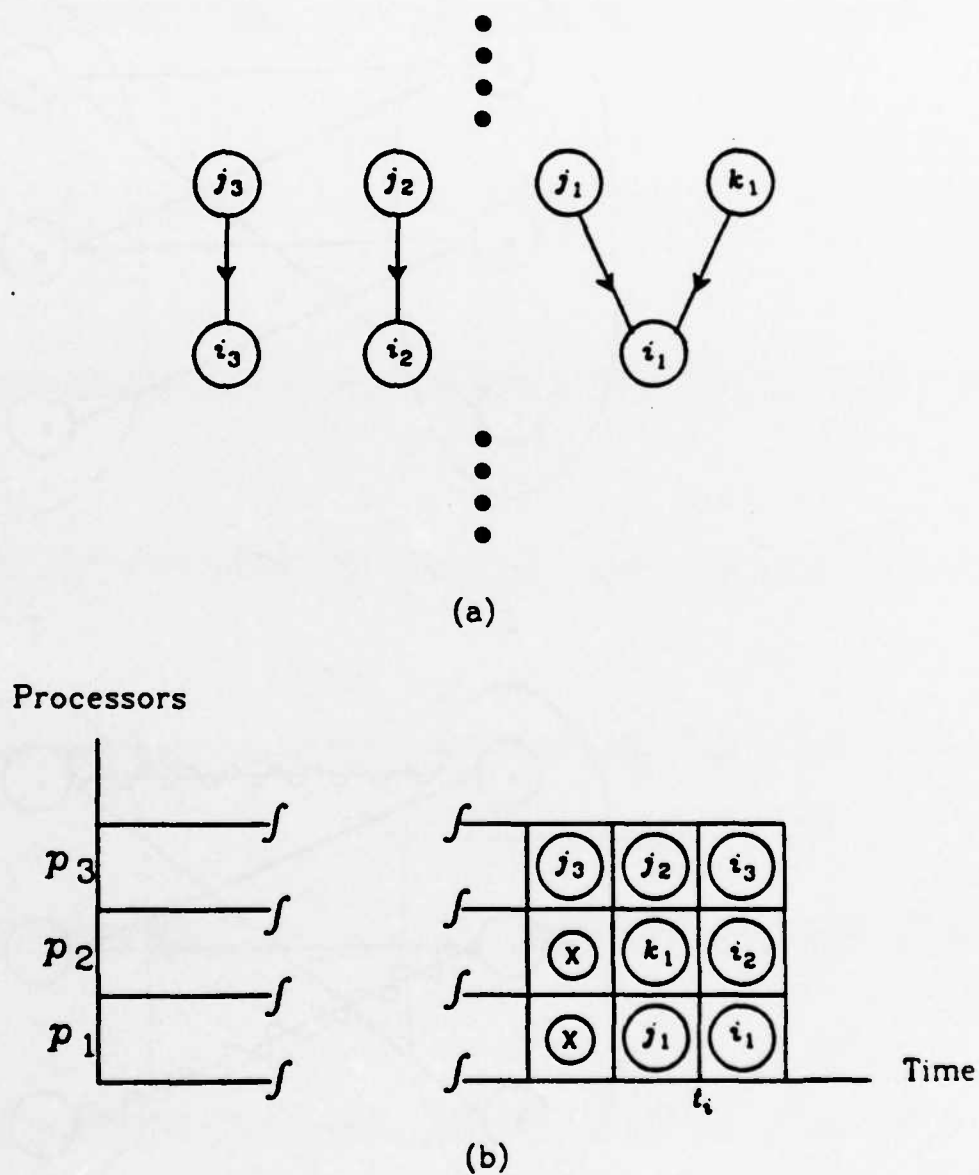
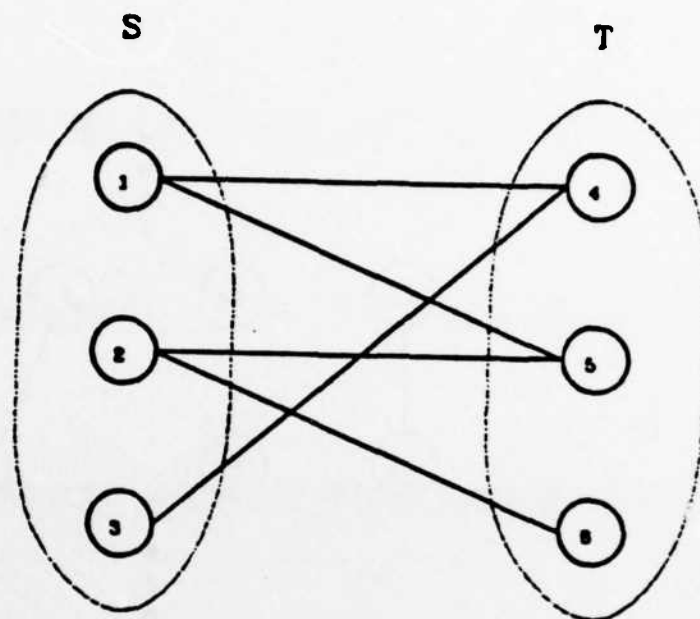
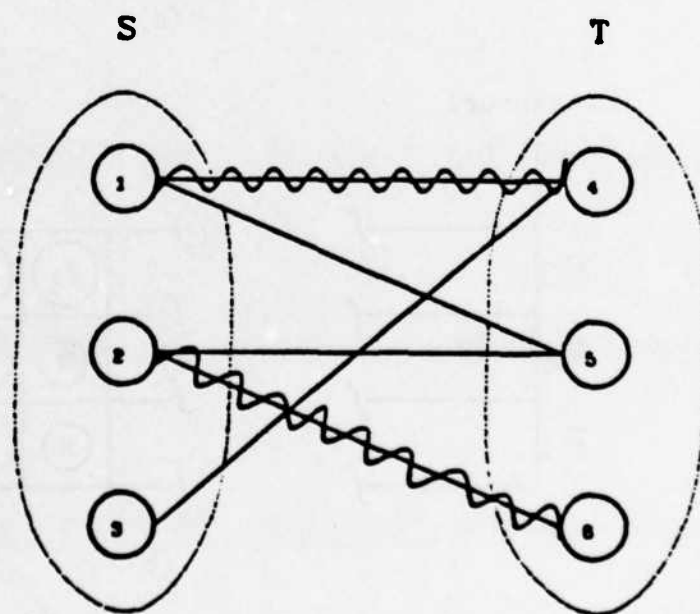


Figure 4.1 (a) Fragment Of An Example Graph. (b) Schedule Showing The Assignment Of Nodes  $i_1, i_2, i_3$  To Processors  $p_1, p_2, p_3$  At Timestep  $t_i$



(a)



(b)

Figure 4.2

(a) An Example Of Bipartite Graph. (b) A Matching In The Bipartite Graph.

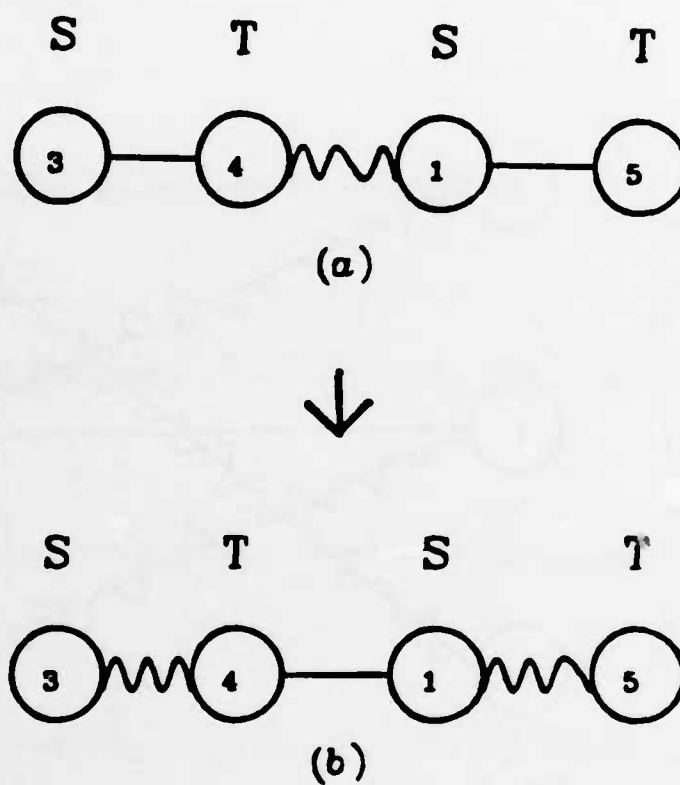


Figure 4.3

(a) An Augmenting Path For The Matching In The Bipartite Graph Of Figure 4.2(a). (b) Excluding  $edge(1,4)$  From And Including  $edge(3,4)$  And  $edge(1,5)$  Into The Matching.

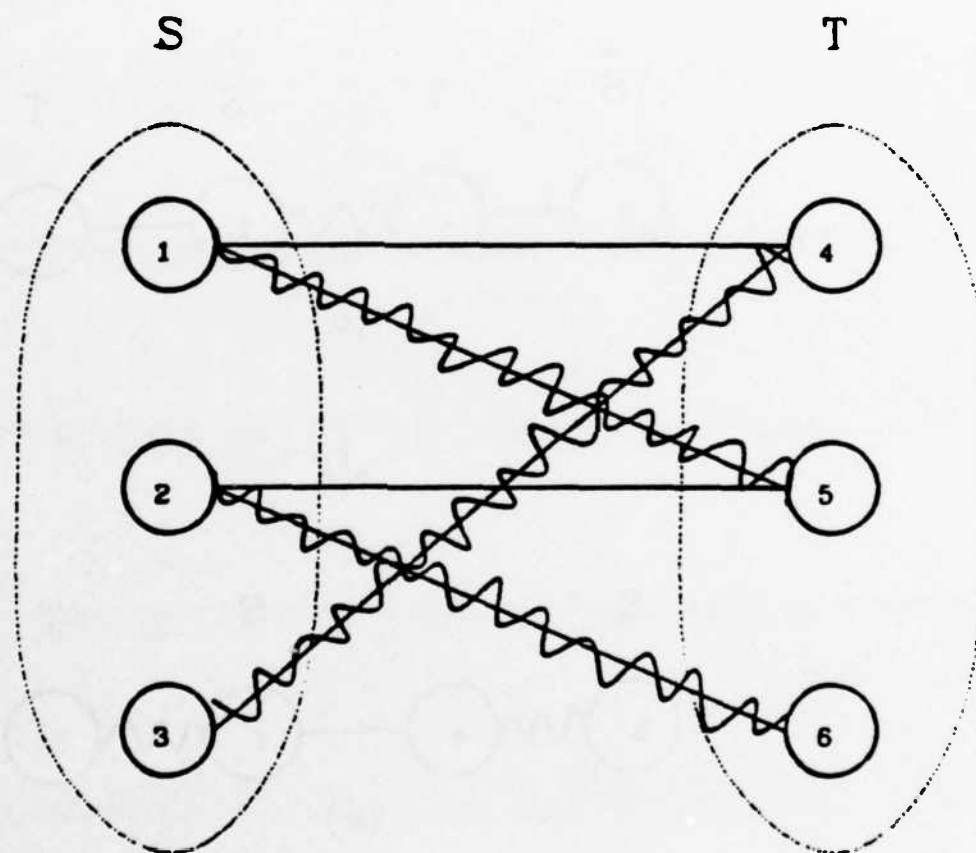
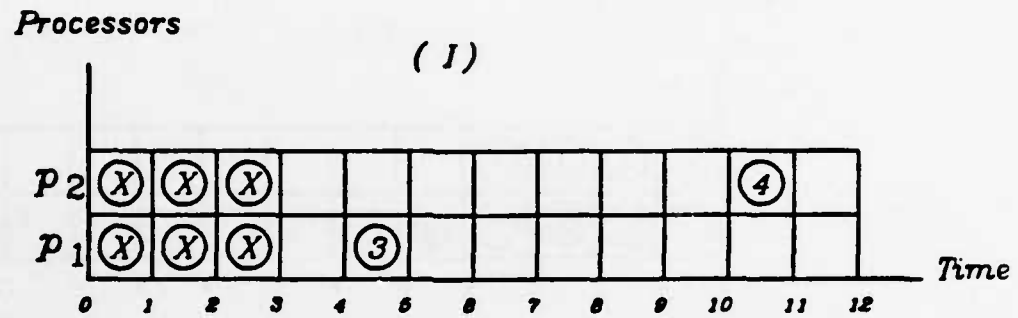
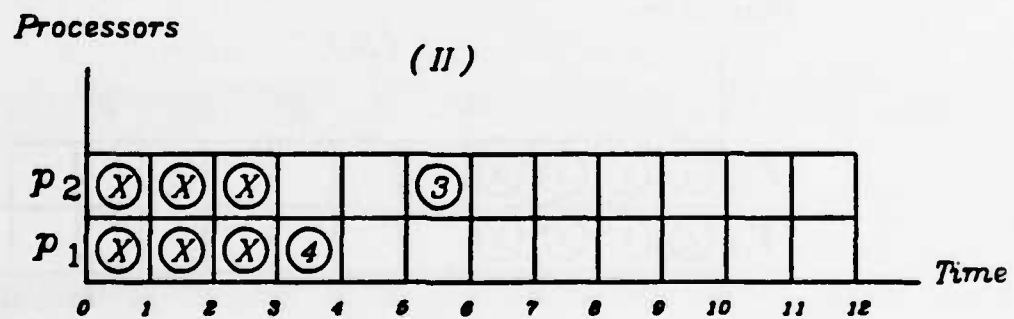


Figure 4.4

The New Matching After Finding The Augmenting Path Of { (3), (4), (1), (5) } In The Matching Shown In Figure 4.2(b).

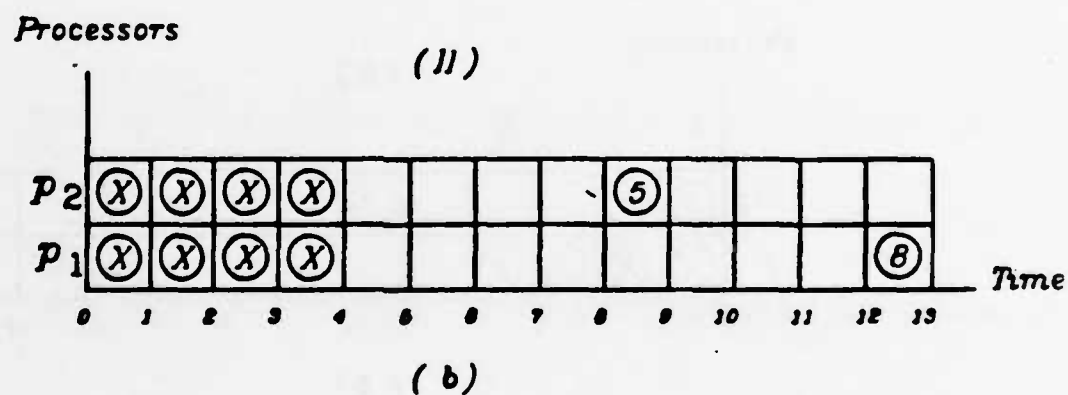
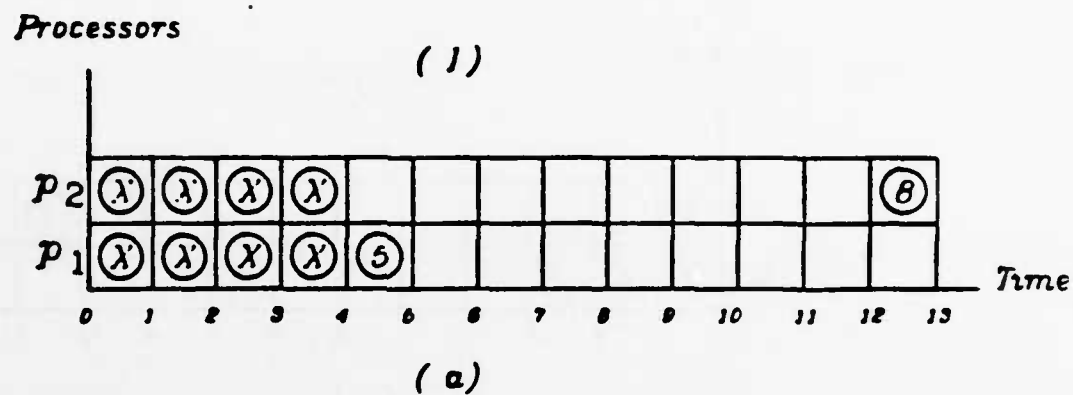


( a )



( b )

**Figure 4.5** Dependency Of The Schedule Obtained On The Order In Which Nodes Are Selected. (a) Node (3) Is Selected First. (b) Node (4) Is Selected First.



**Figure 4.6** Two Possible Schedules With Same Partial Completion Time. Nodes Can Be Assigned To Processors Earlier In The Later Timestep After Node (5) in (a) Than in (b).

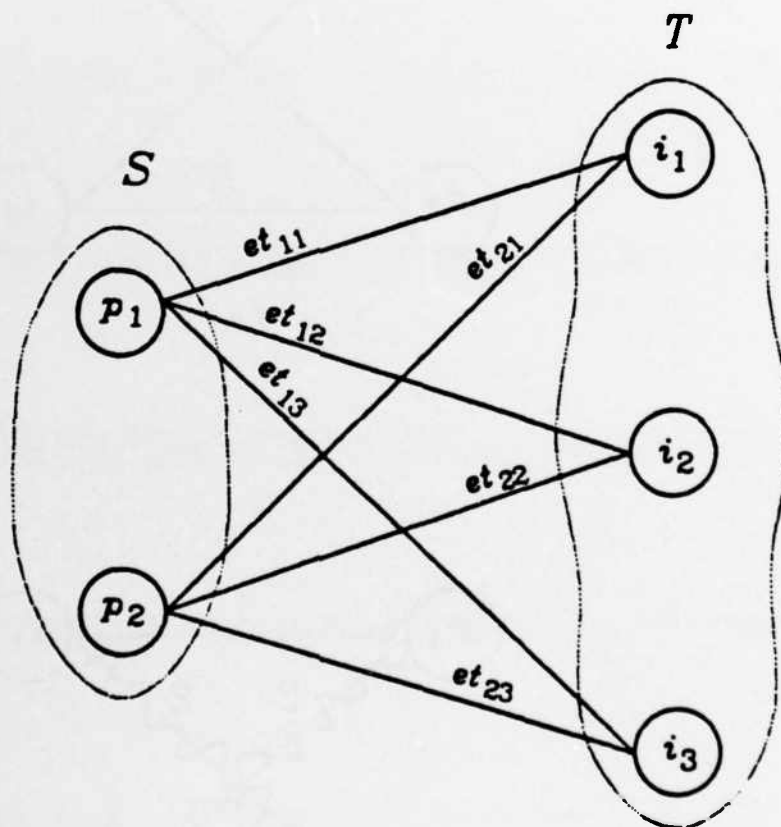
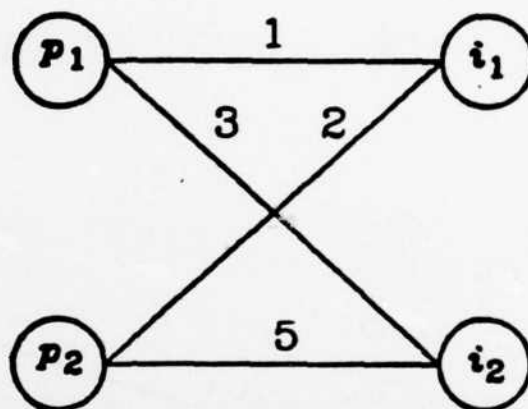


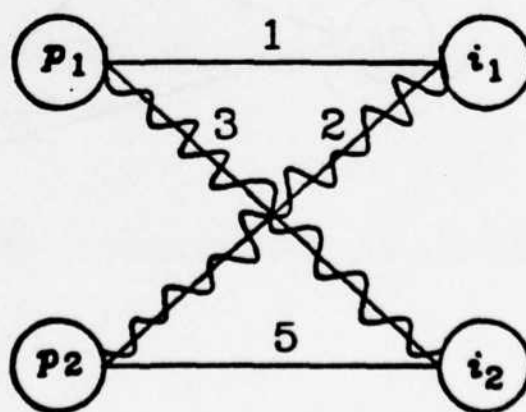
Figure 4.7

A Weighted Bipartite Graph Representation Of The Scheduling Environment Of Two Processors  $p_1, p_2$  And Three Nodes  $i_1, i_2, i_3$ .





(a)



(b)

Figure 4.8

(a) Weighted Bipartite Graph Representation Of The Scheduling Environment Of Two Processors  $p_1, p_2$  and Two Nodes  $i_1, i_2$ . (b) The Matching Obtained By Min\_Max Matching Algorithm.

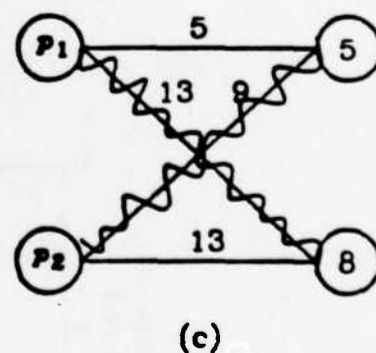
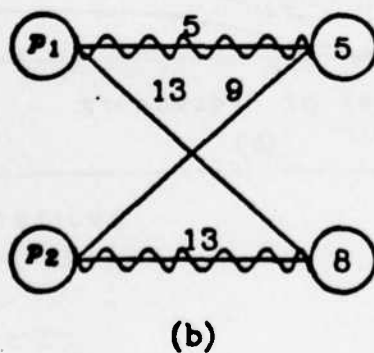
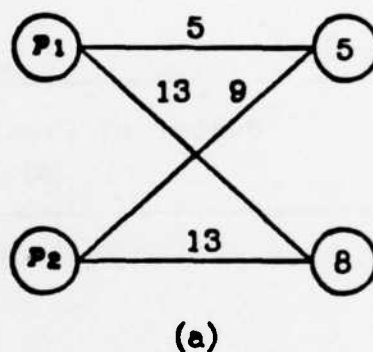


Figure 4.9

(a) Weighted Bipartite Graph Representation Of The Scheduling Environment Of Two Processors  $p_1$ ,  $p_2$  And Two Nodes (5), (8). The Min\_Max Matching Gives Either (b) Or (c). Weighted Matching Applied To The Modified Weights On The Graph Of (a) Will Result In (b) Only.

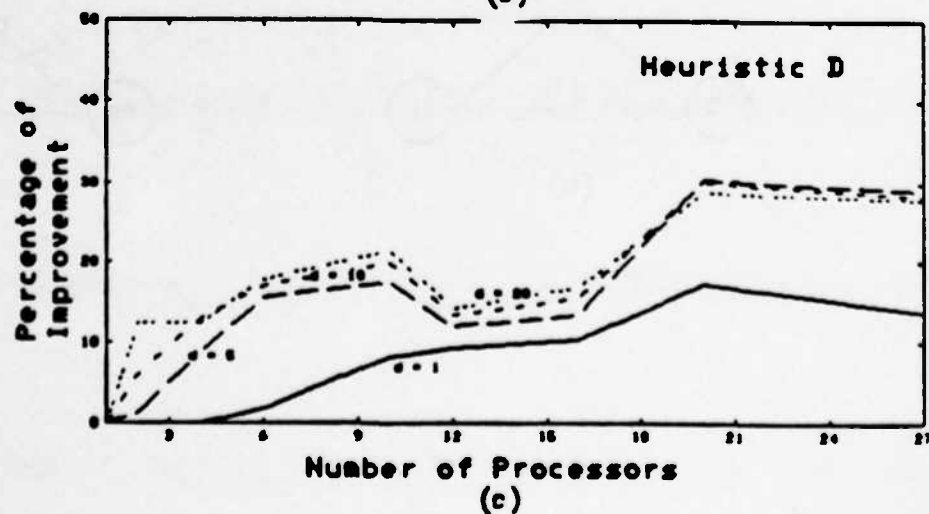
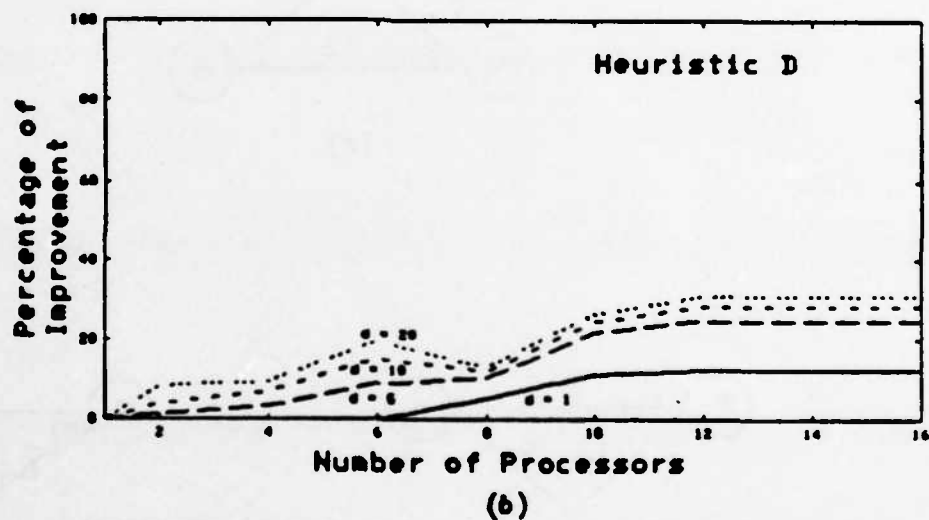
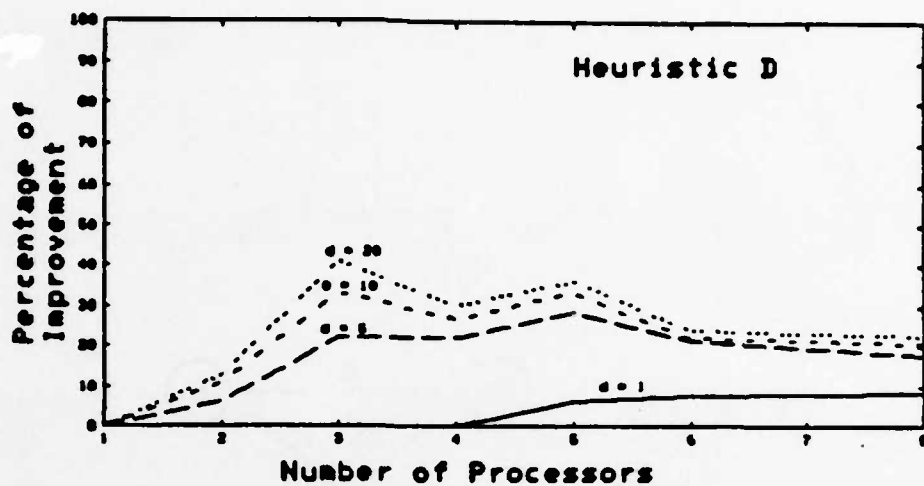
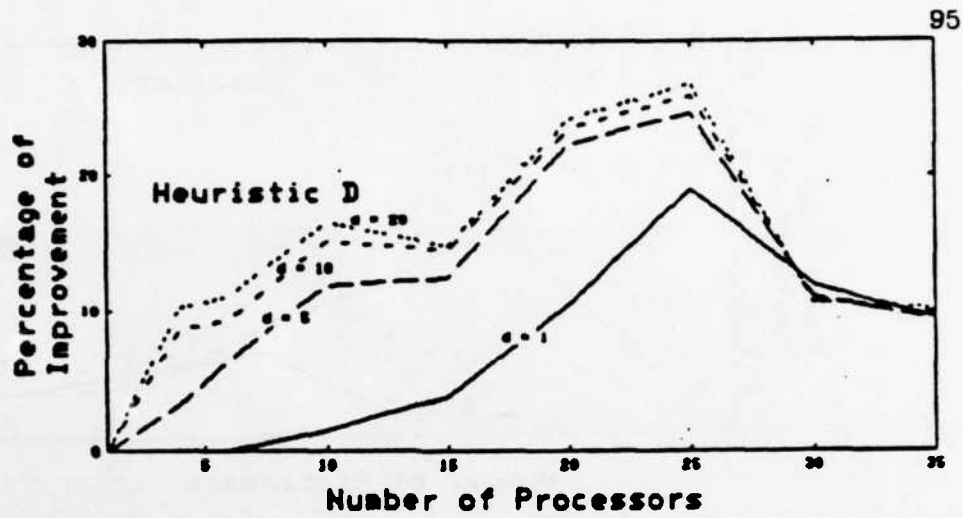
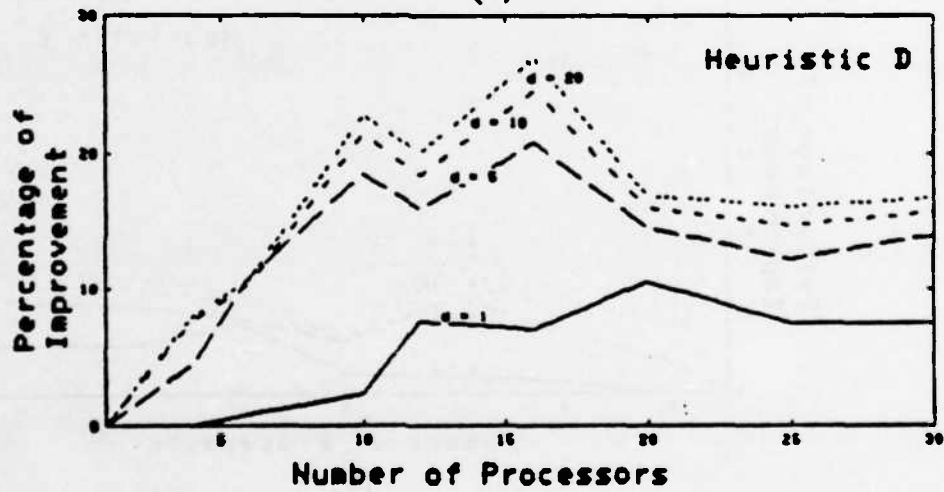


Figure 4.10

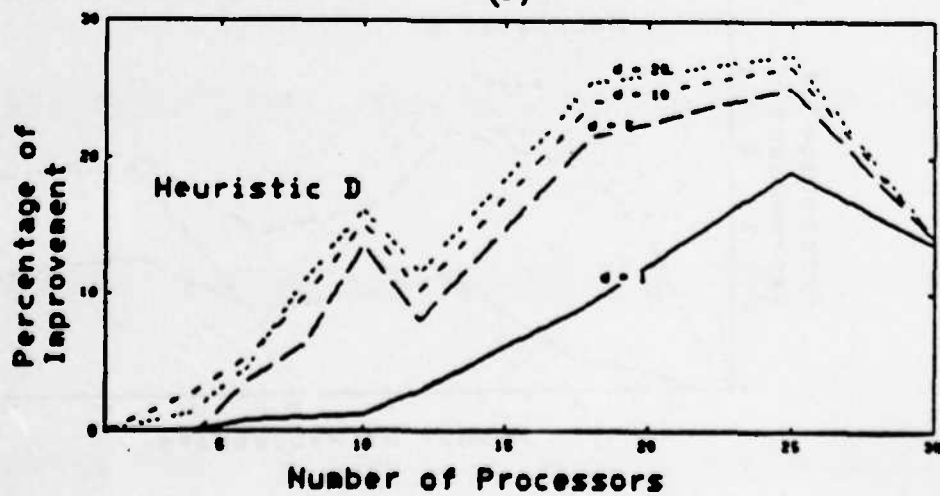
Percentage Of Speedup Improvement of Heuristic D Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 58 Nodes (b) 128 Nodes (c) 295 Nodes



(a)

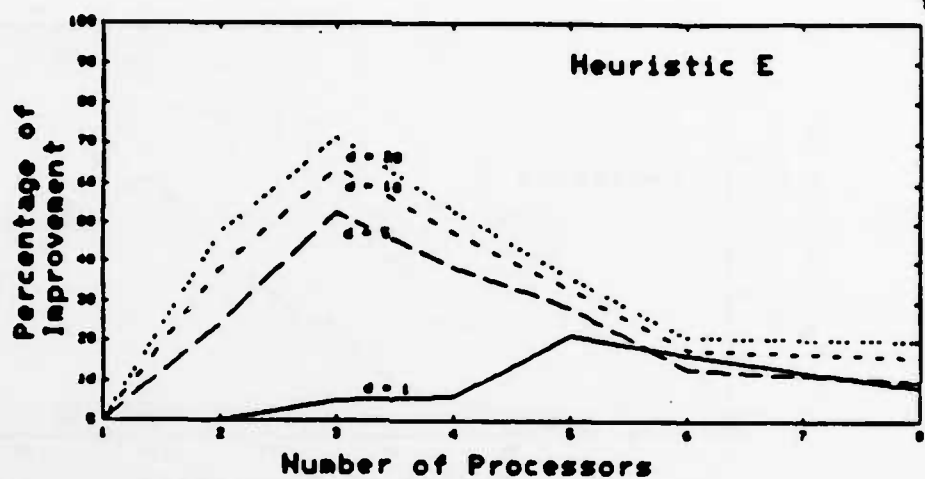


(b)

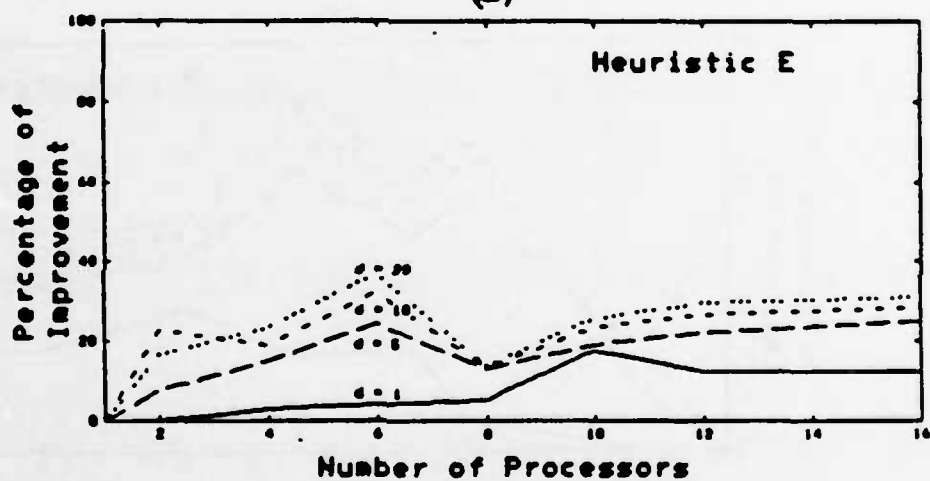


(c)

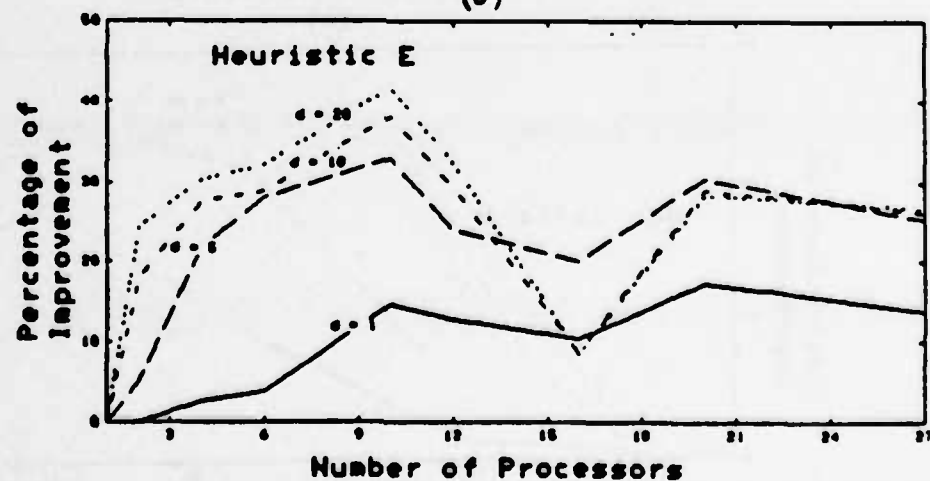
Figure 4.11 Percentage Of Speedup Improvement of Heuristic D Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 844 Nodes (b) 725 Nodes (c) 767 Nodes



(a)



(b)



(c)

Figure 4.12 Percentage Of Speedup Improvement of Heuristic E Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 58 Nodes (b) 128 Nodes (c) 295 Nodes

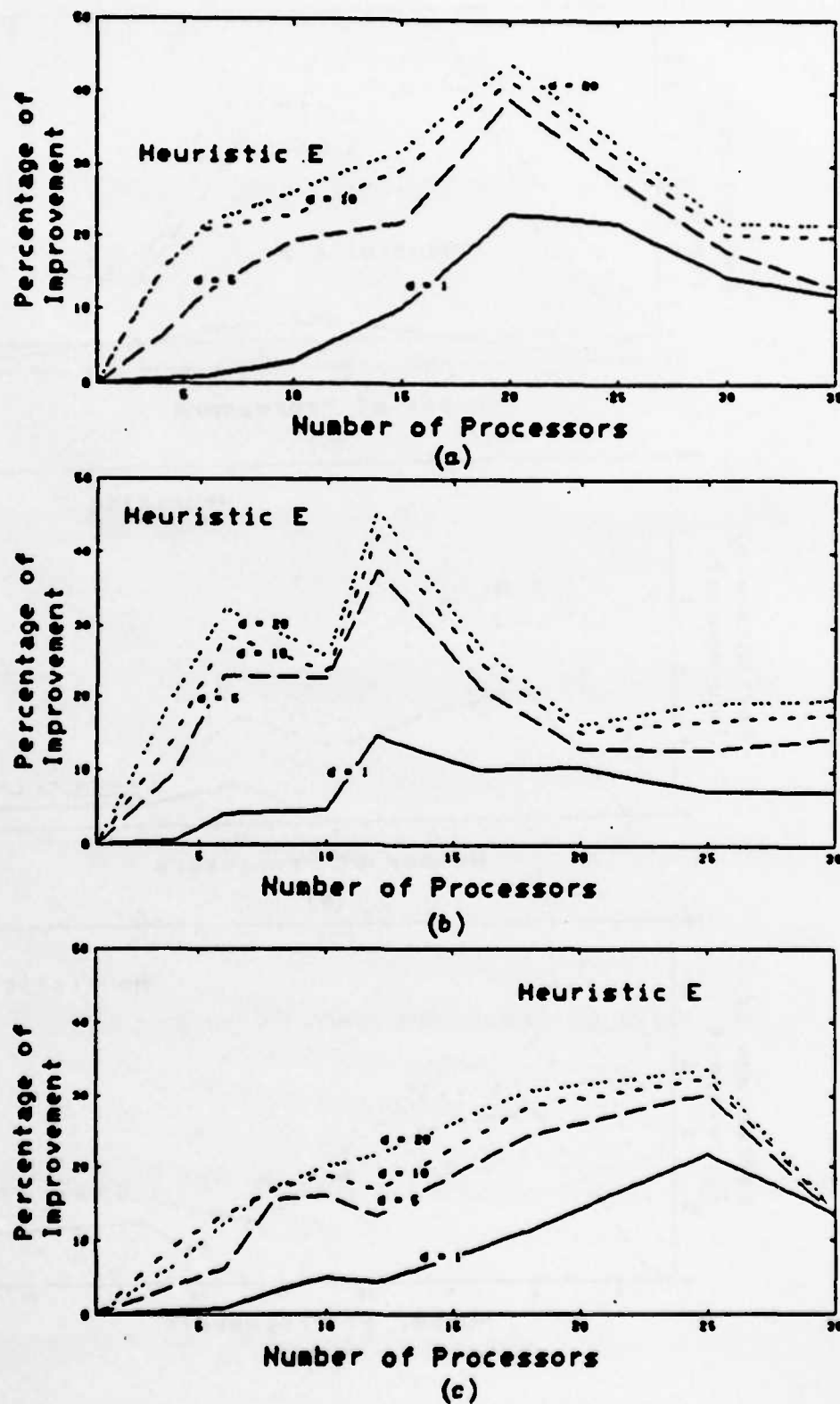


Figure 4.13 Percentage Of Speedup Improvement of Heuristic E Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 844 Nodes (b) 725 Nodes (c) 787 Nodes

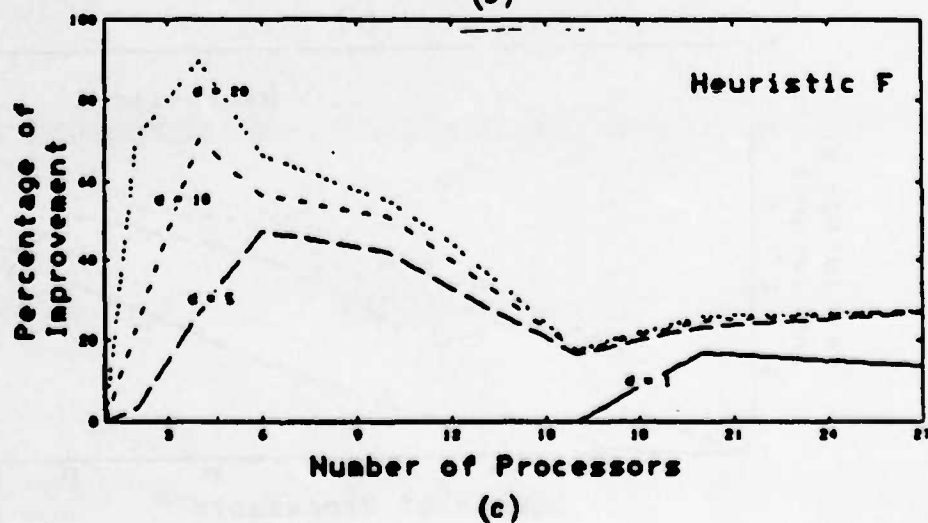
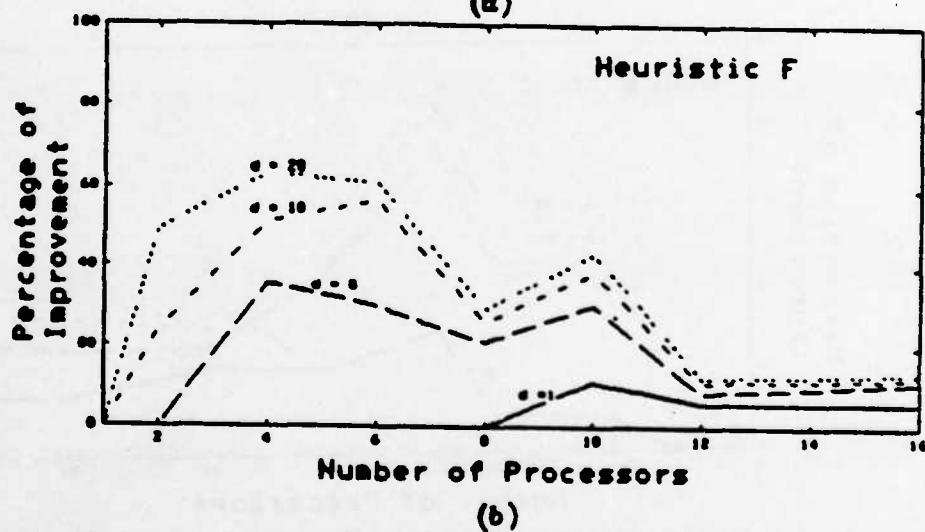
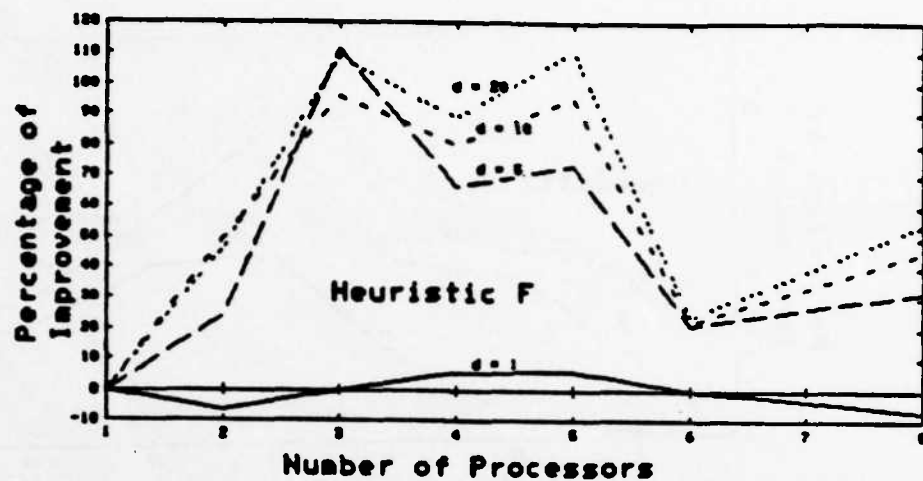


Figure 4.14 Percentage Of Speedup Improvement of Heuristic F Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 58 Nodes (b) 128 Nodes (c) 295 Nodes



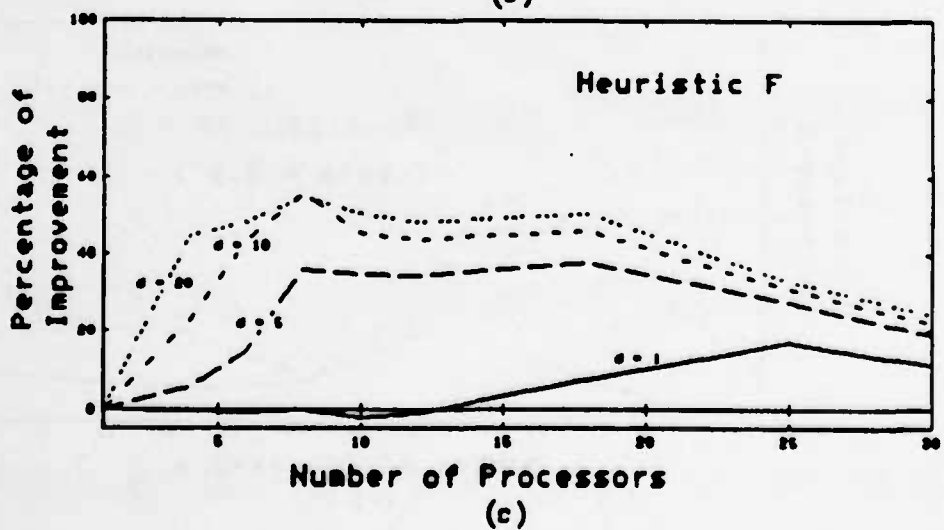
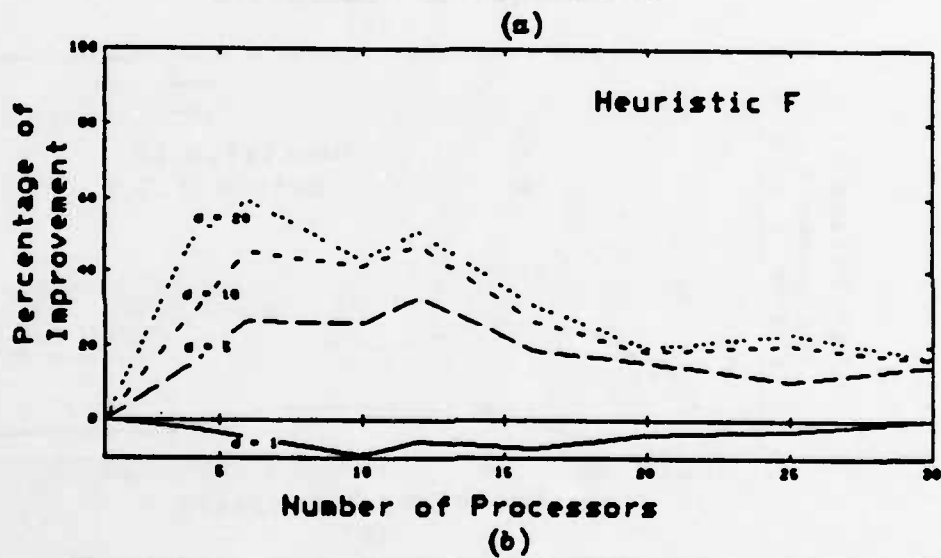
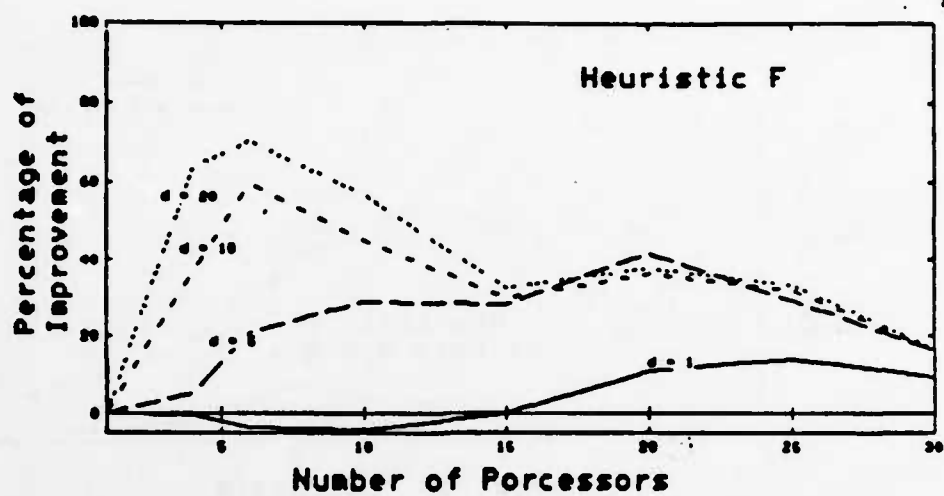


Figure 4.15 Percentage Of Speedup Improvement of Heuristic F Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 644 Nodes (b) 725 Nodes (c) 767 Nodes

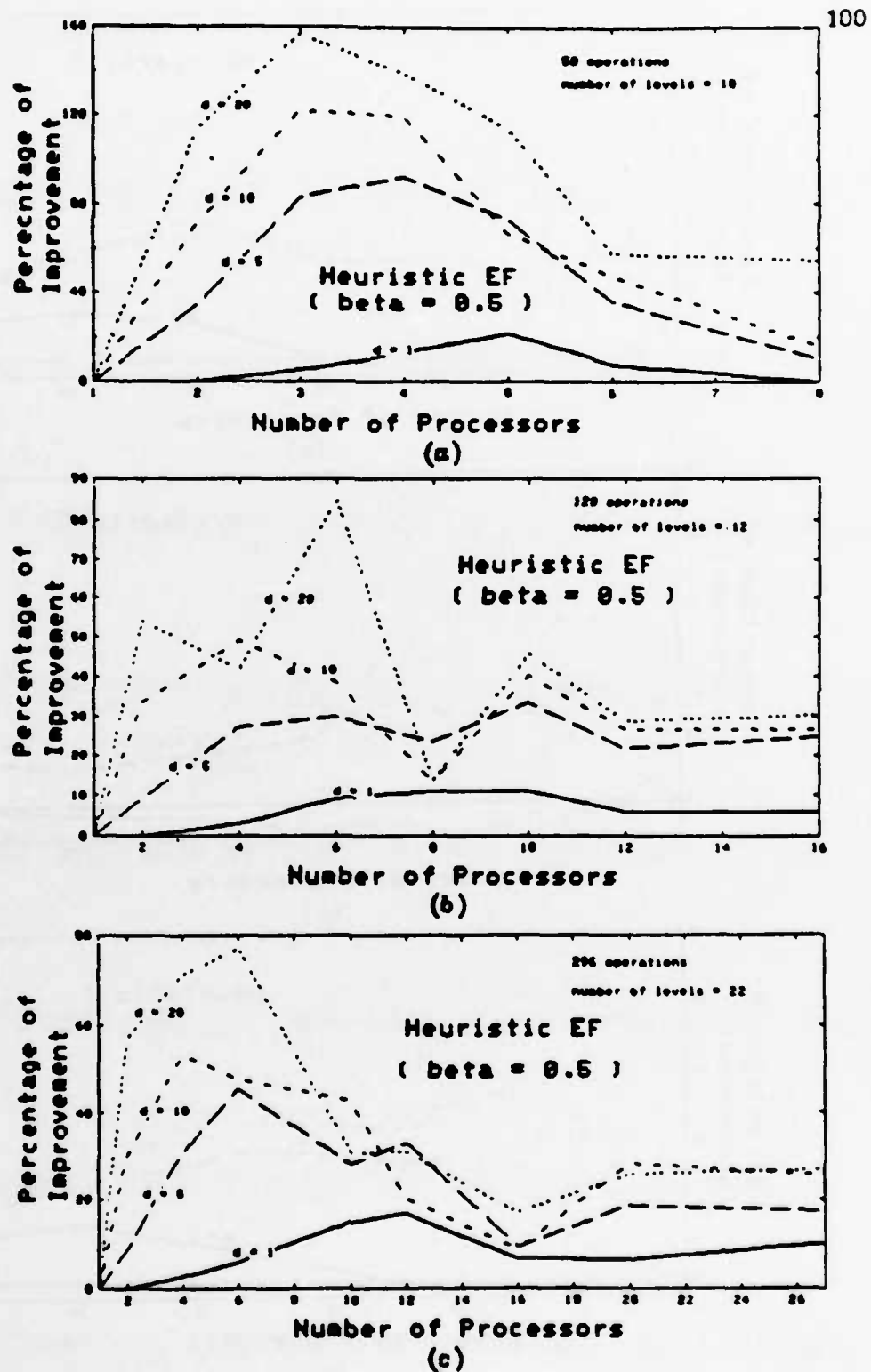


Figure 4.18 Percentage Of Speedup Improvement of Heuristic EF ( $\beta = 0.5$ ) Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 58 Nodes (b) 128 Nodes (c) 295 Nodes

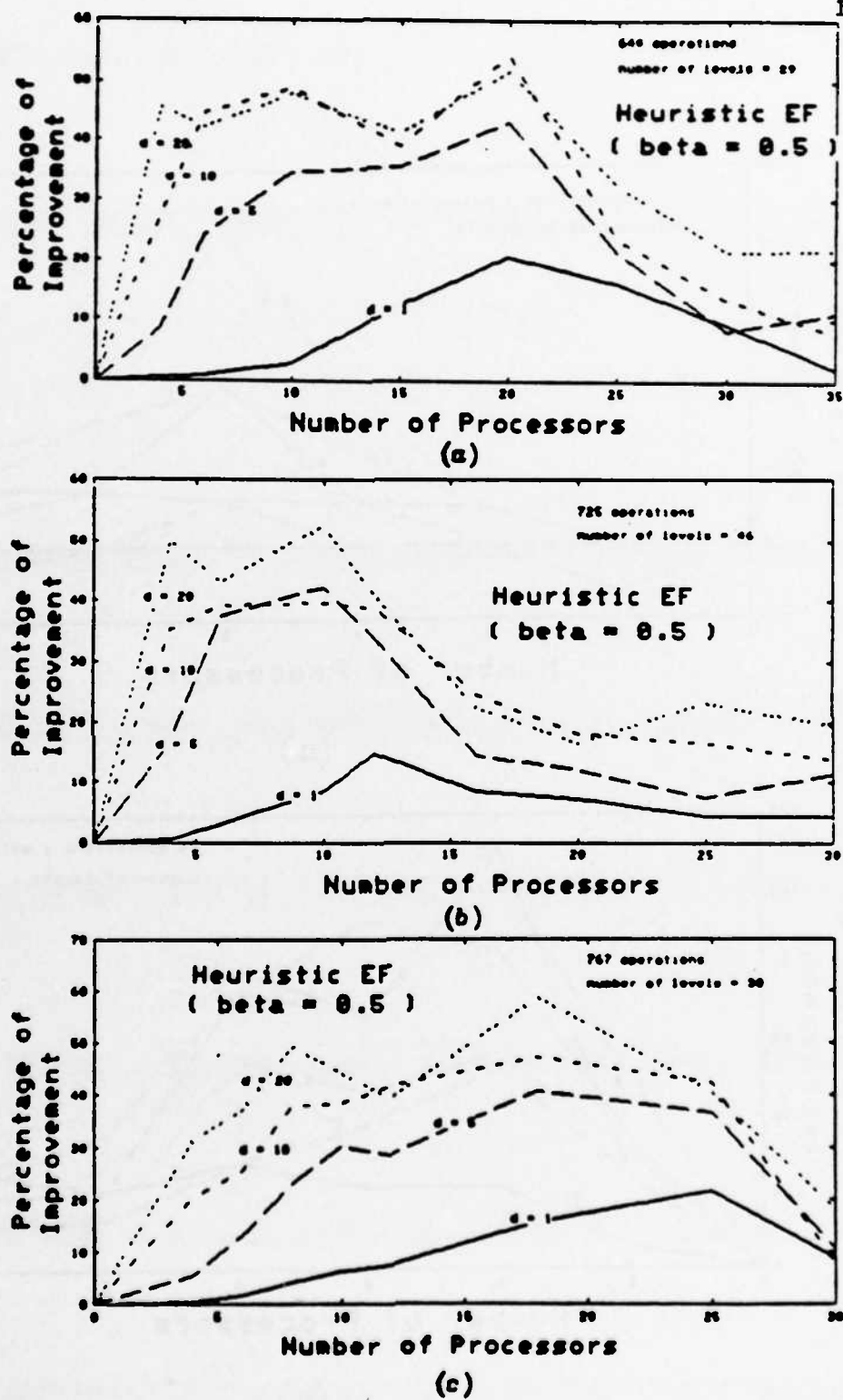
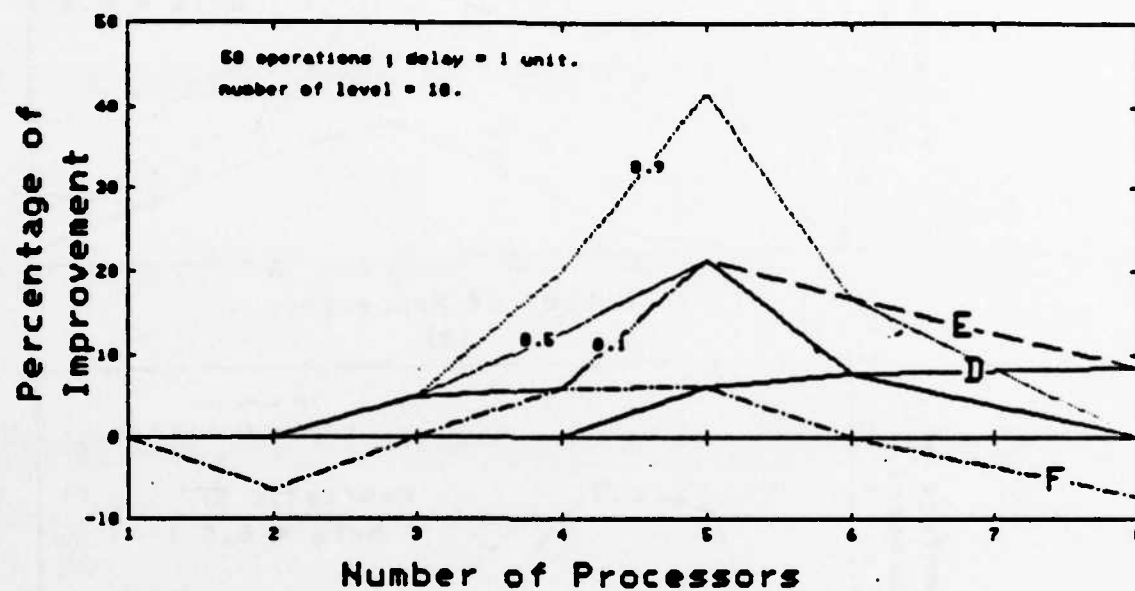
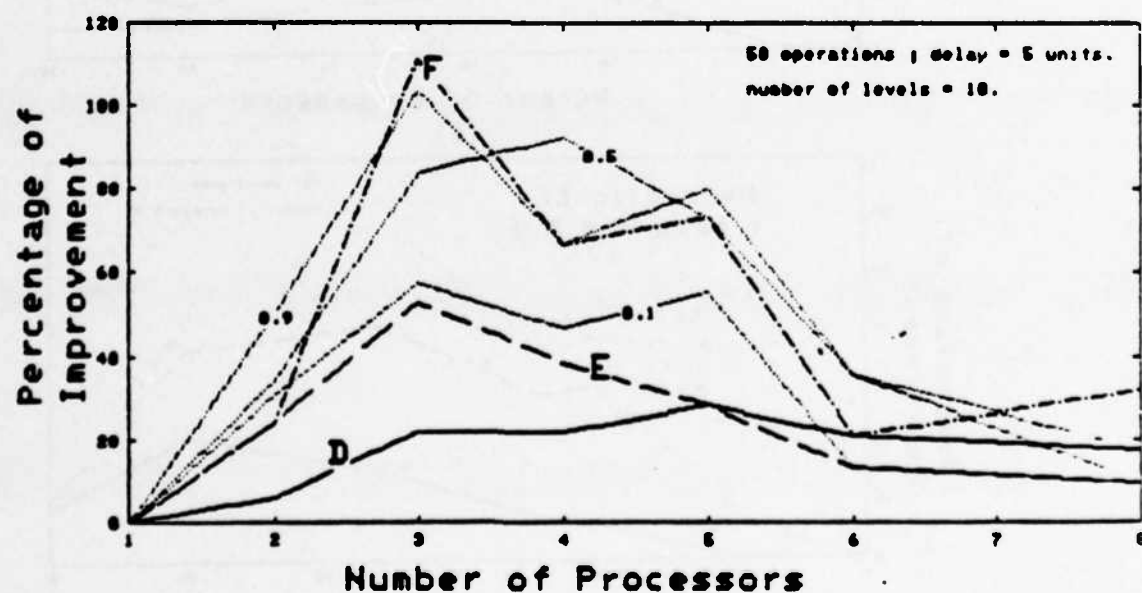


Figure 4.17 Percentage Of Speedup Improvement of Heuristic EF ( $\beta = 0.5$ ) Over Hu's Level Scheduling With Different Constant Delays For The Following Task Graphs. (a) 644 Nodes (b) 725 Nodes (c) 767 Nodes



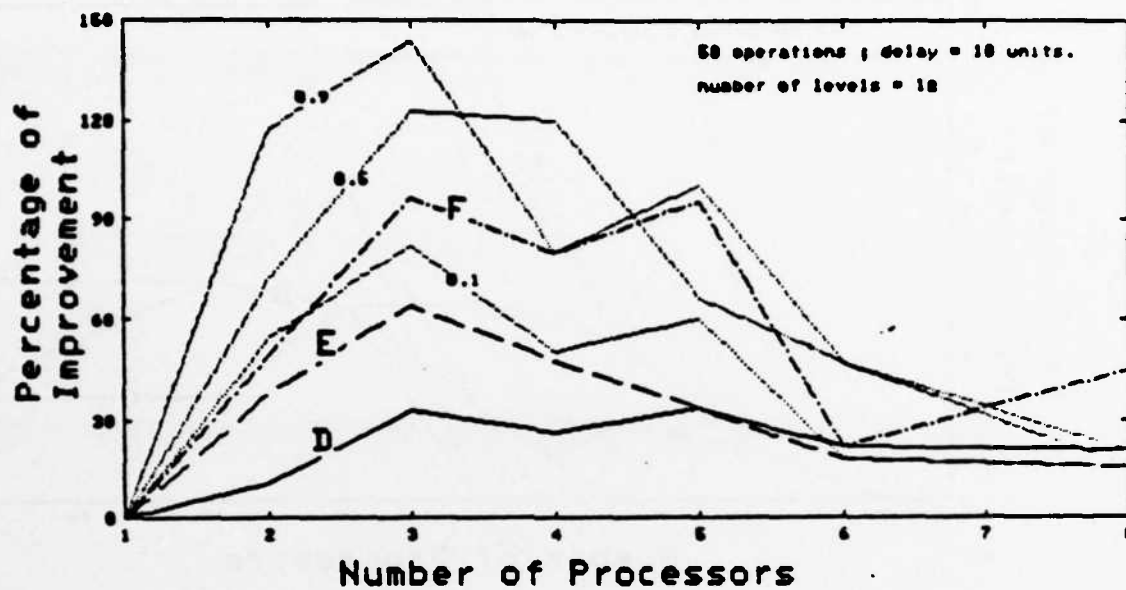
(a)



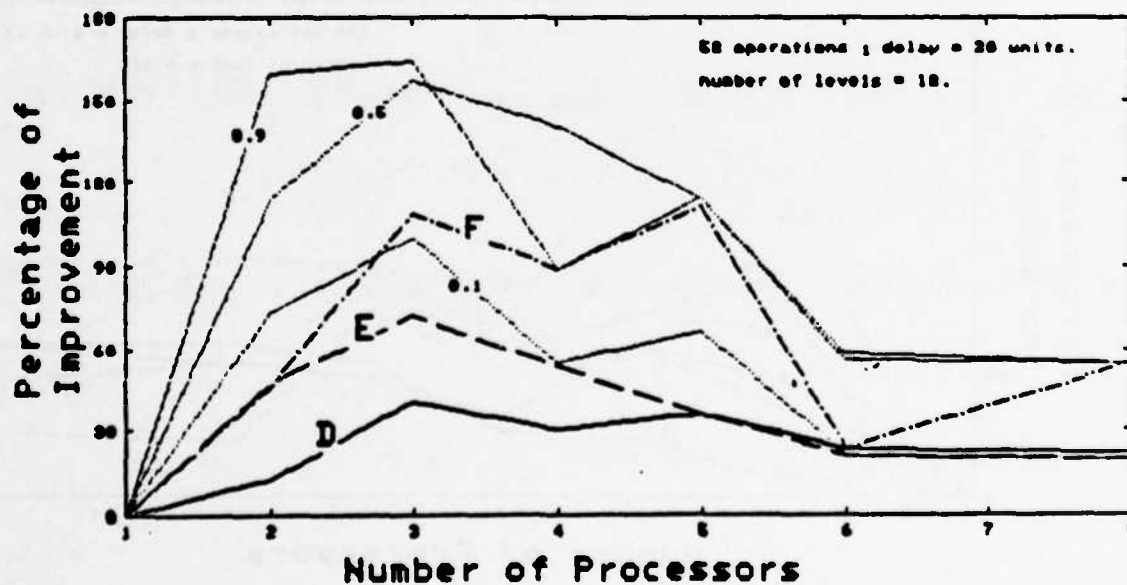
(b)

Figure 4.18

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 58 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.



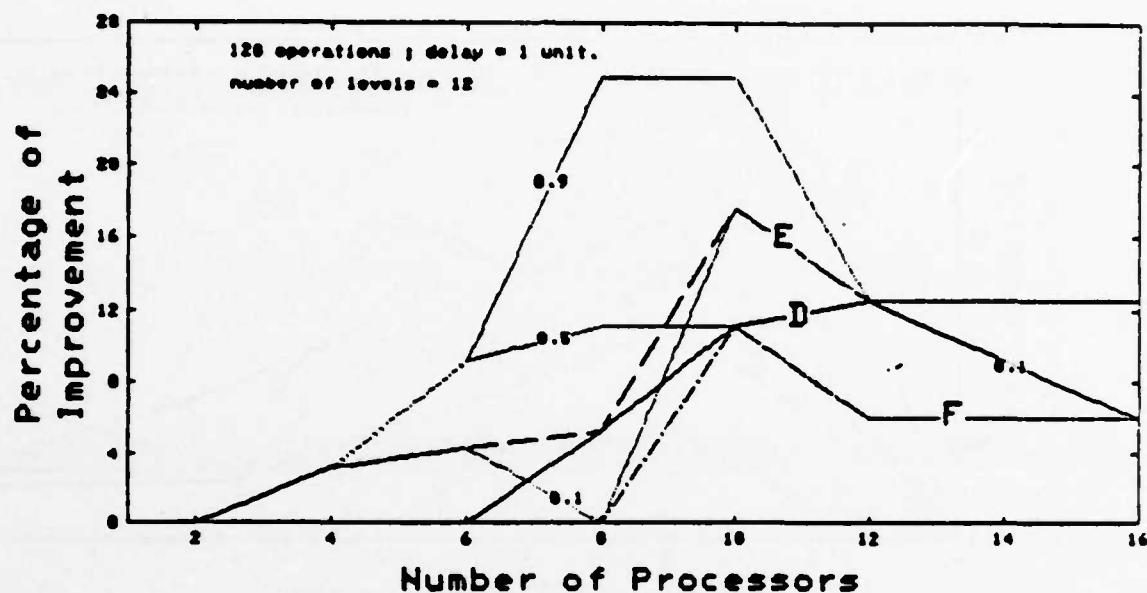
(a)



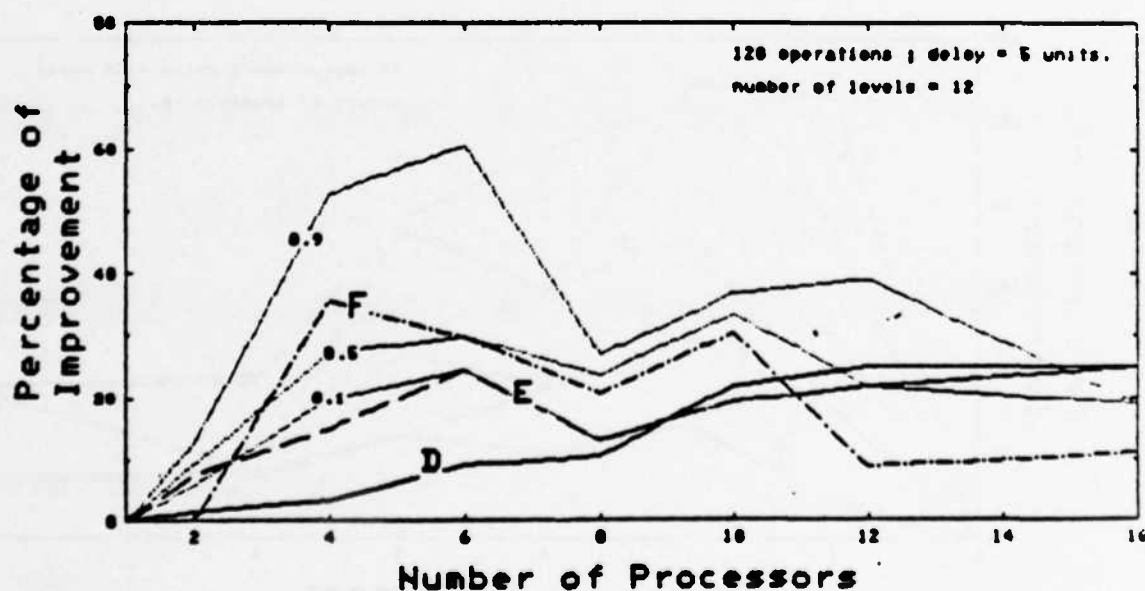
(b)

Figure 4.19

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 58 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.



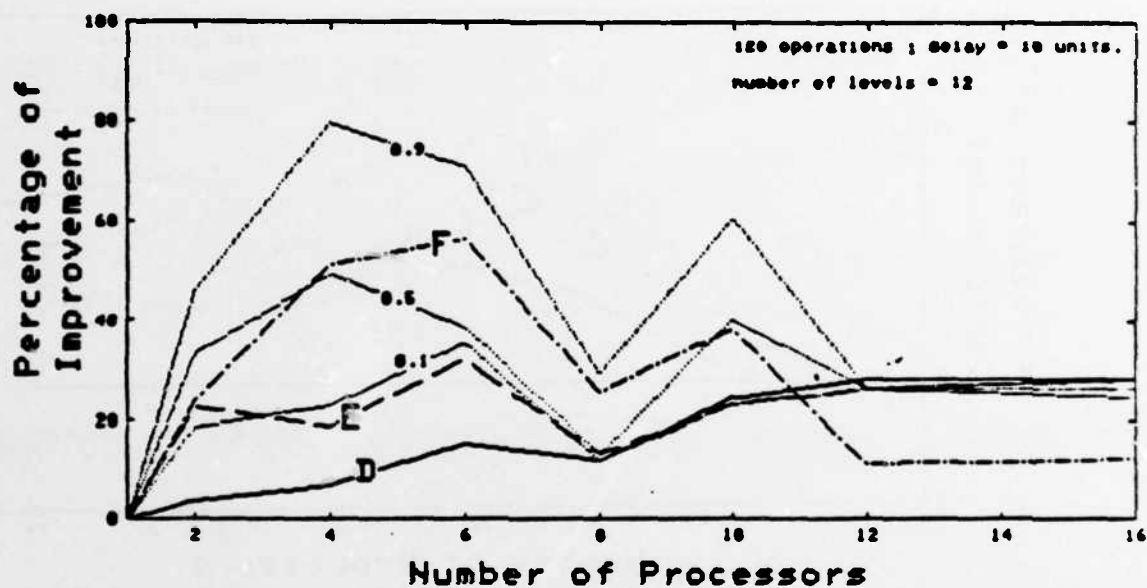
(a)



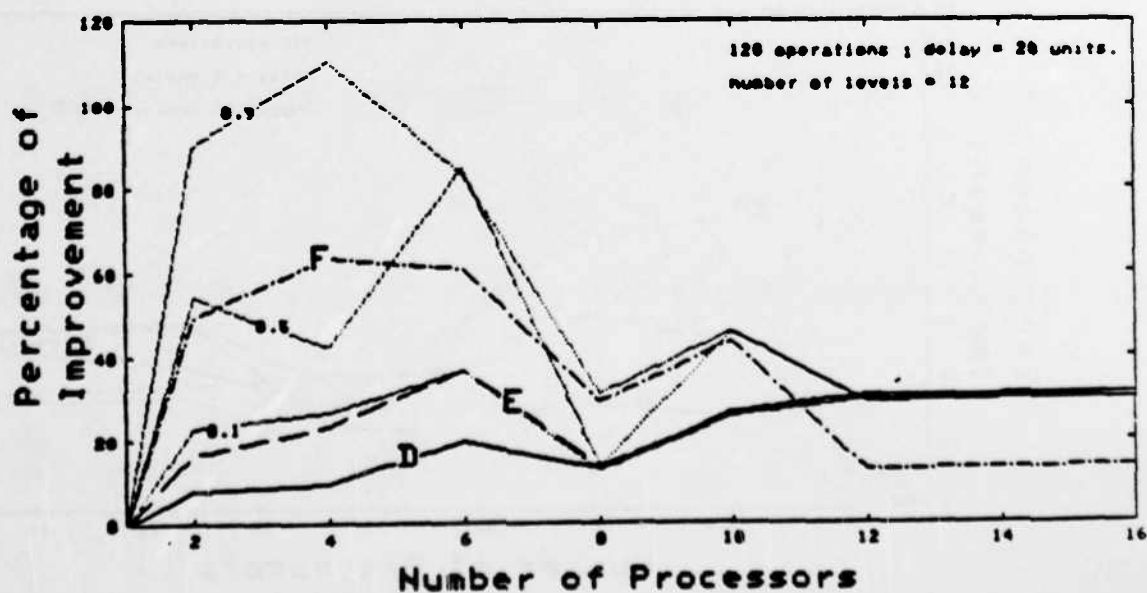
(b)

Figure 4.20

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 128 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.



(a)



(b)

Figure 4.21

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 128 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.



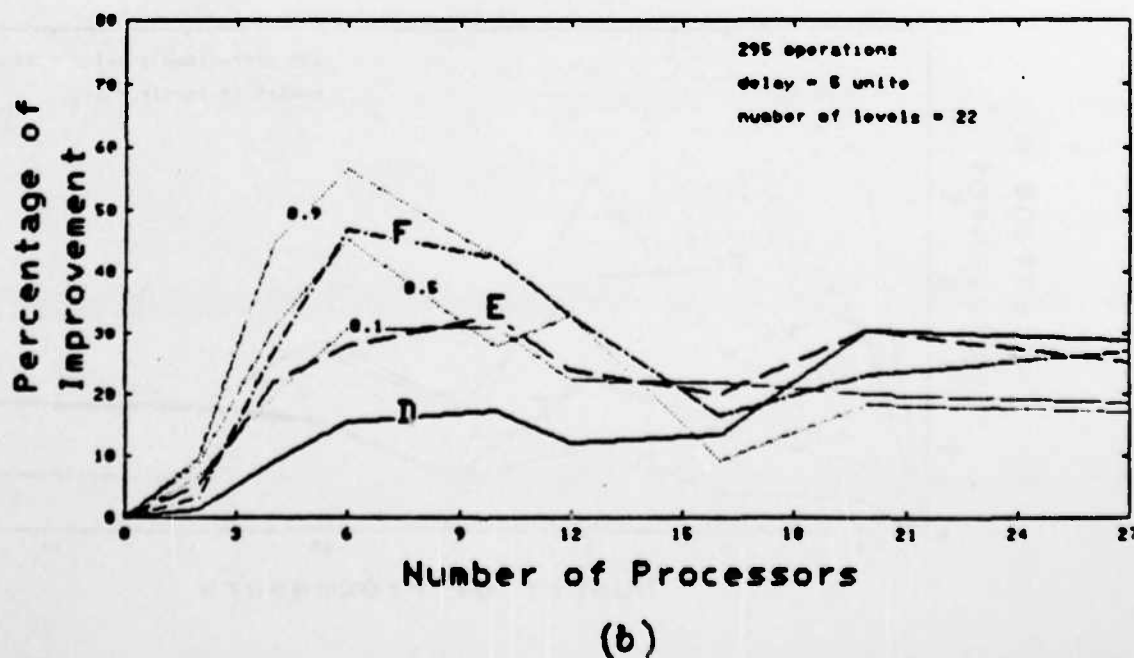
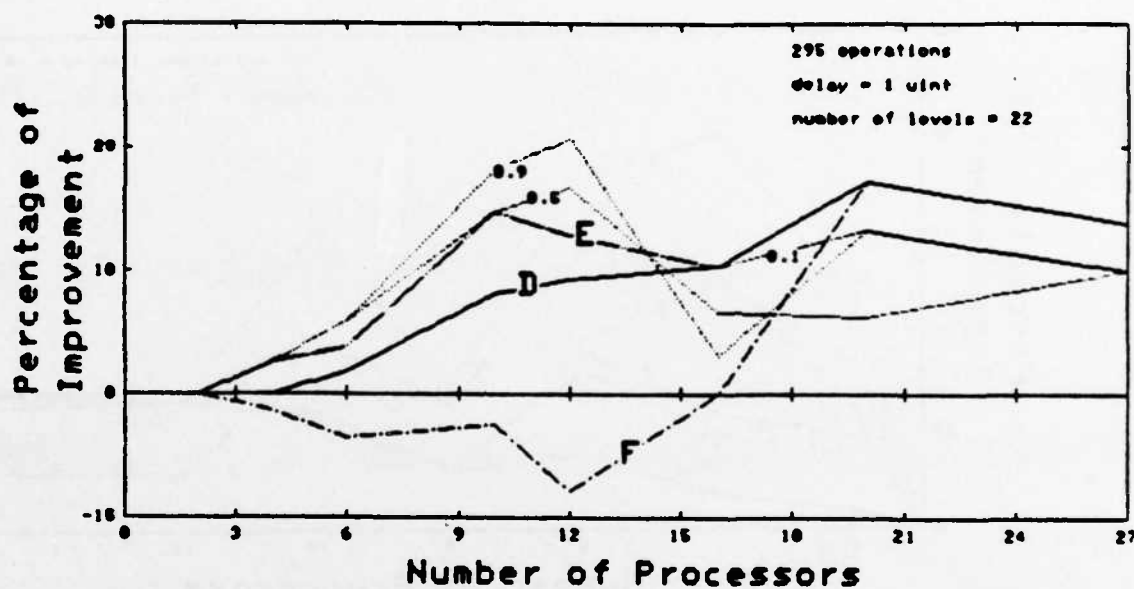
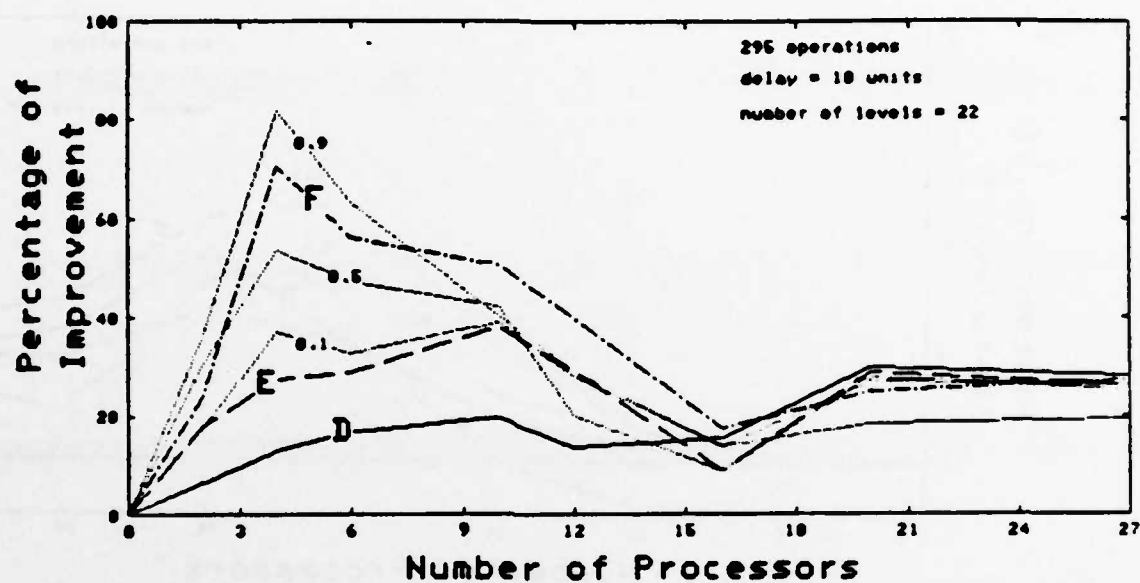
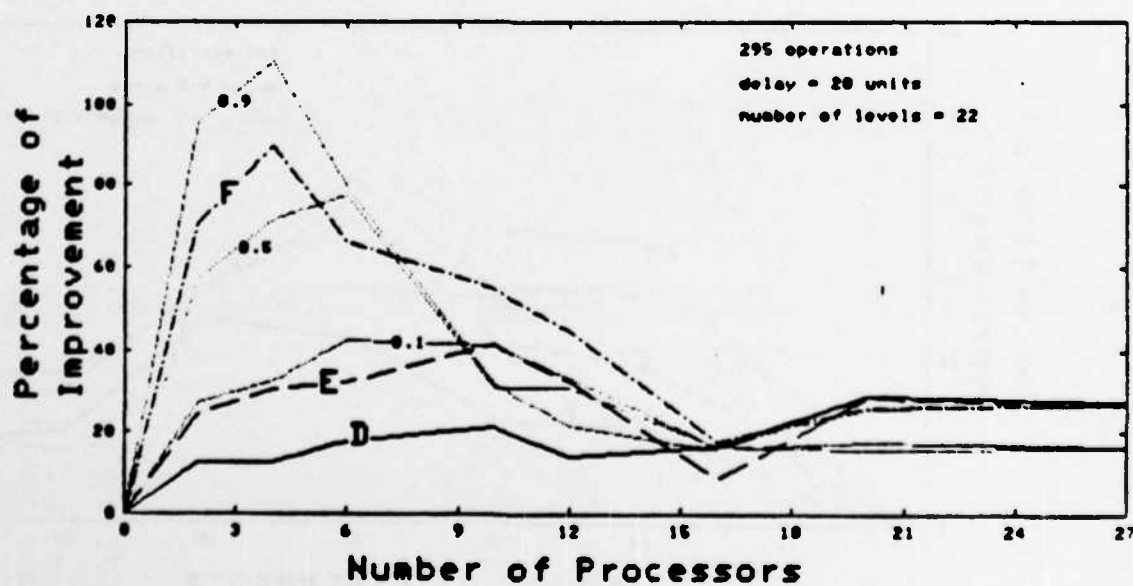


Figure 4.22

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 295 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.



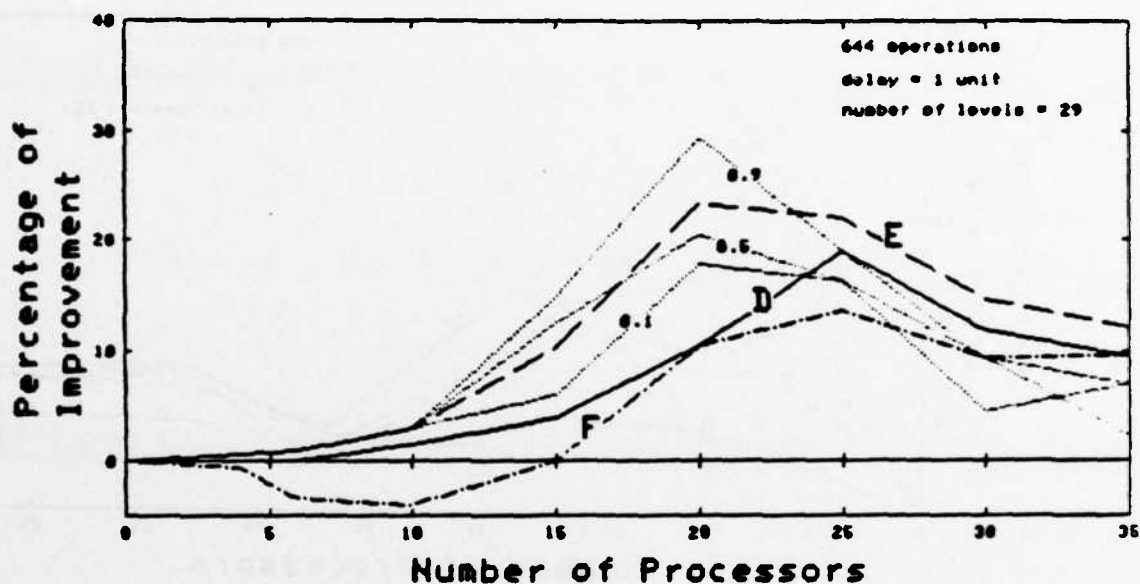
(a)



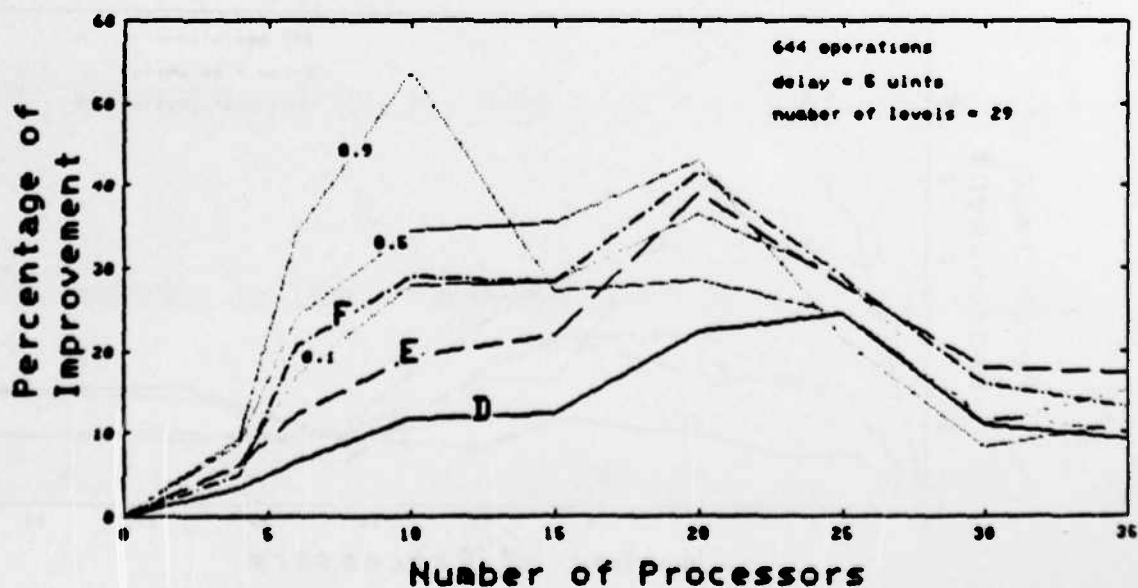
(b)

Figure 4.23

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 295 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.



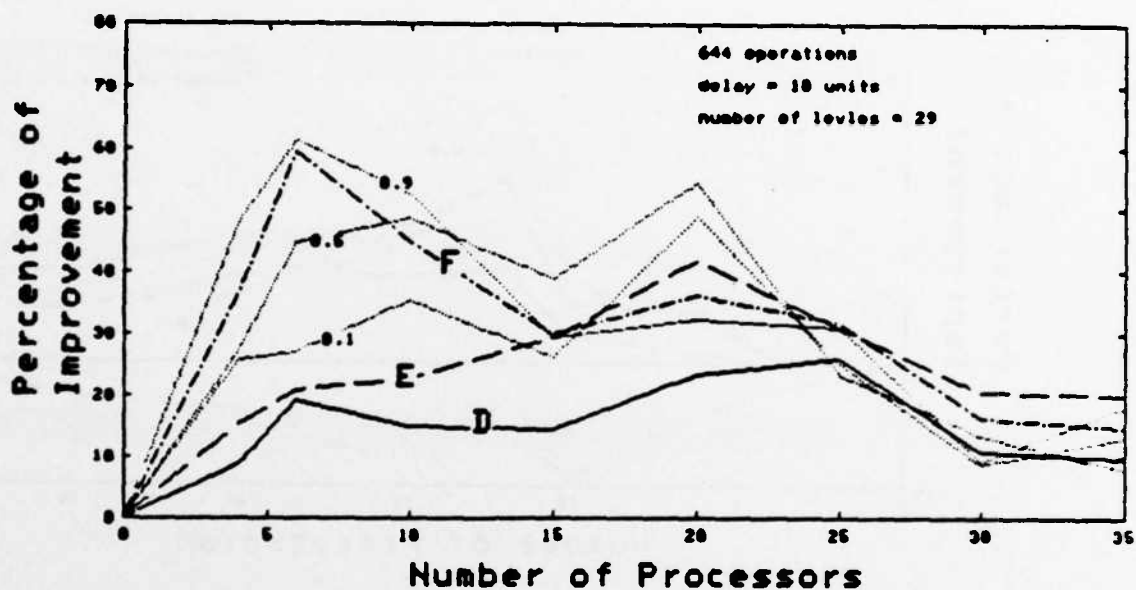
(a)



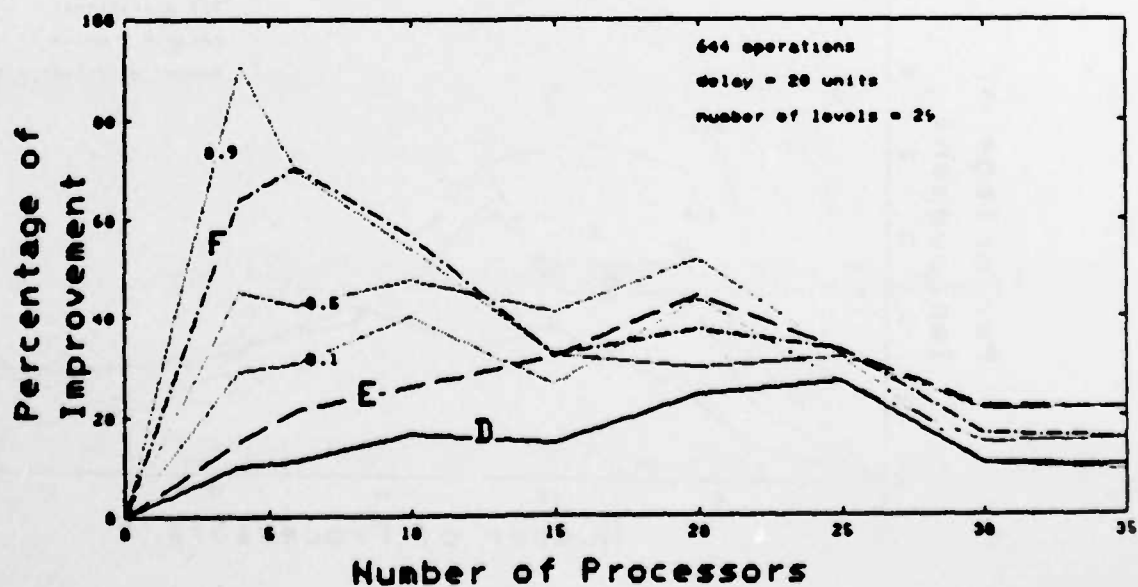
(b)

Figure 4.24

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 644 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.



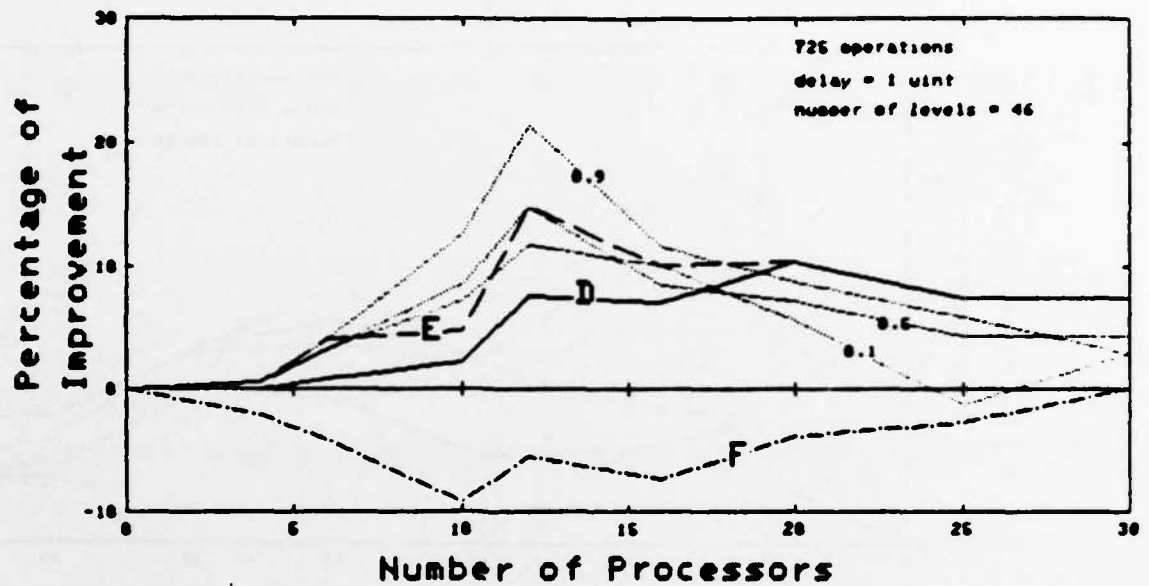
(a)



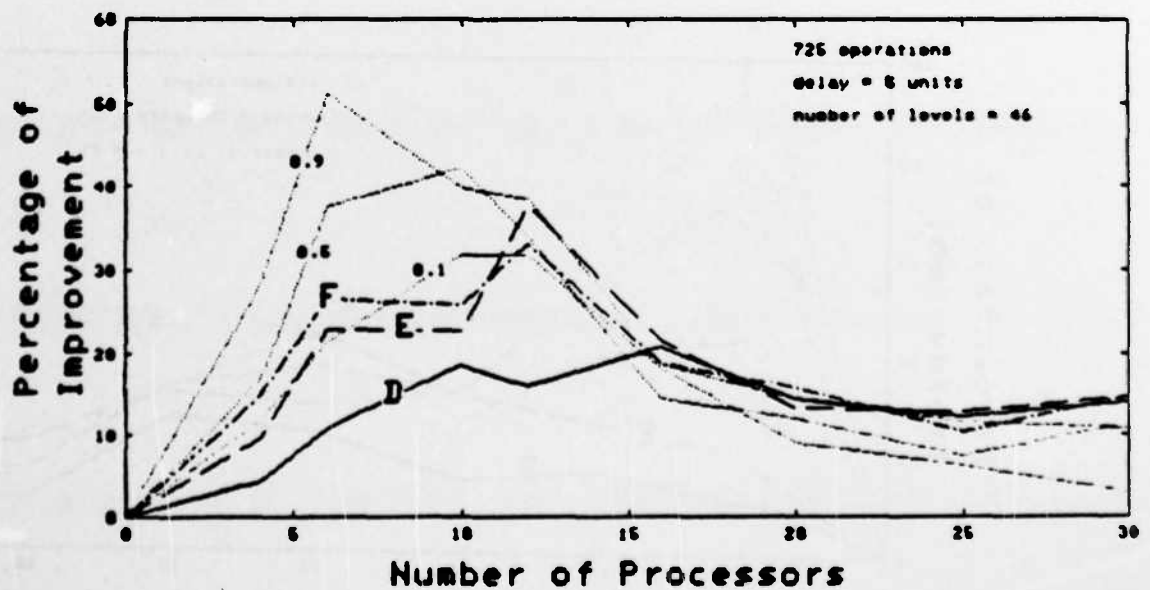
(b)

Figure 4.25

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 844 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.



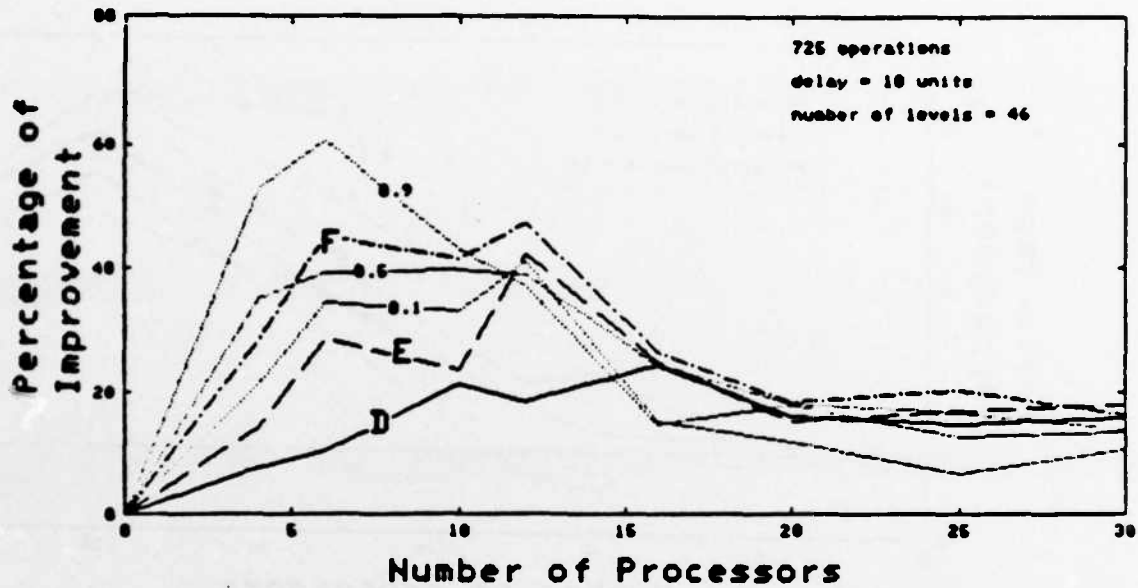
(a)



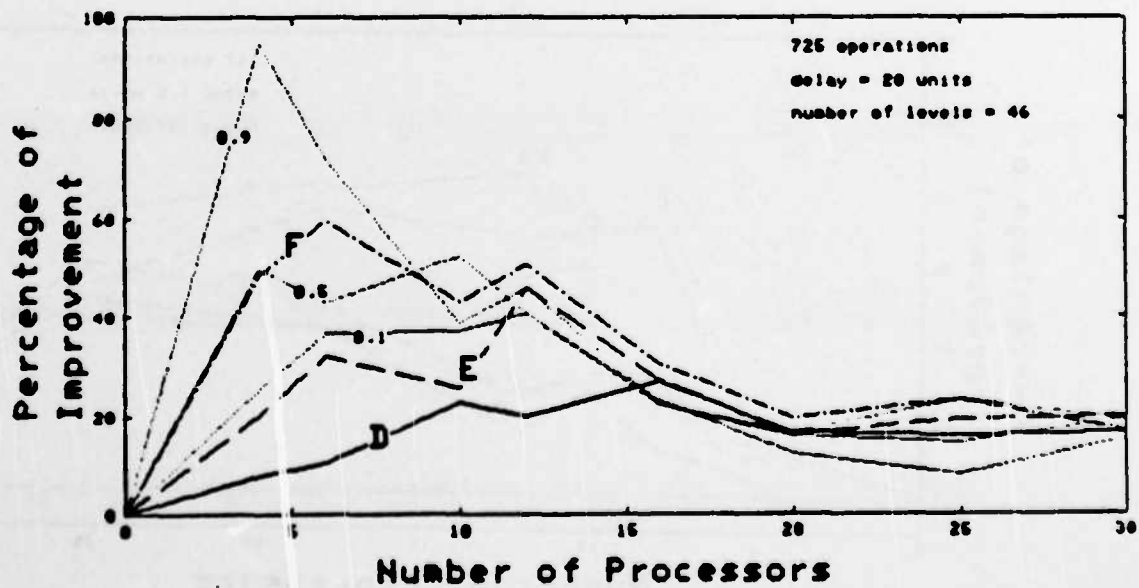
(b)

Figure 4.26

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 725 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.



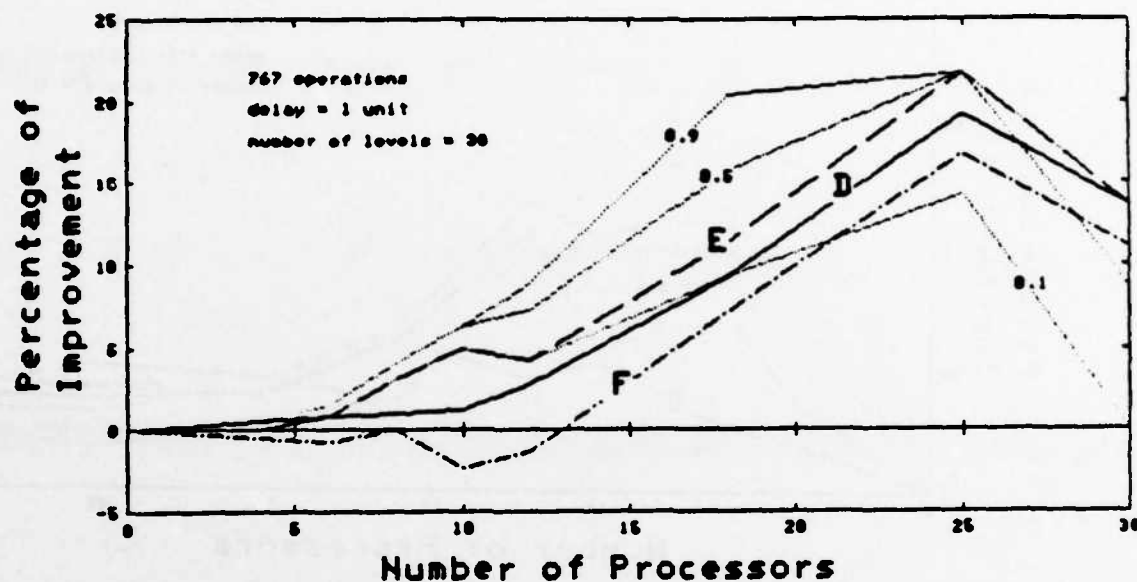
(a)



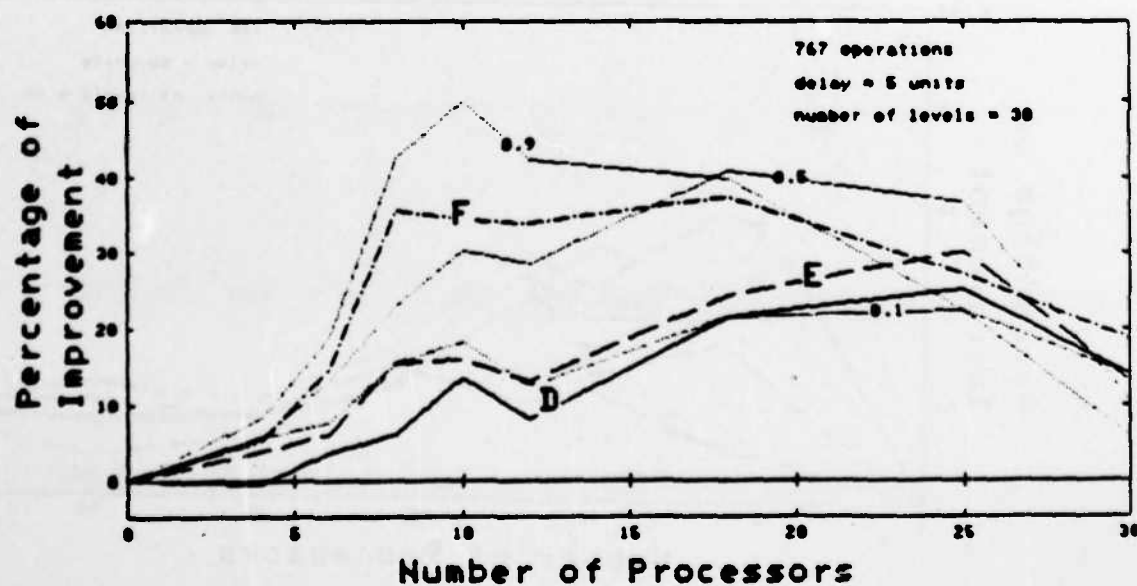
(b)

Figure 4.27

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 725 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.



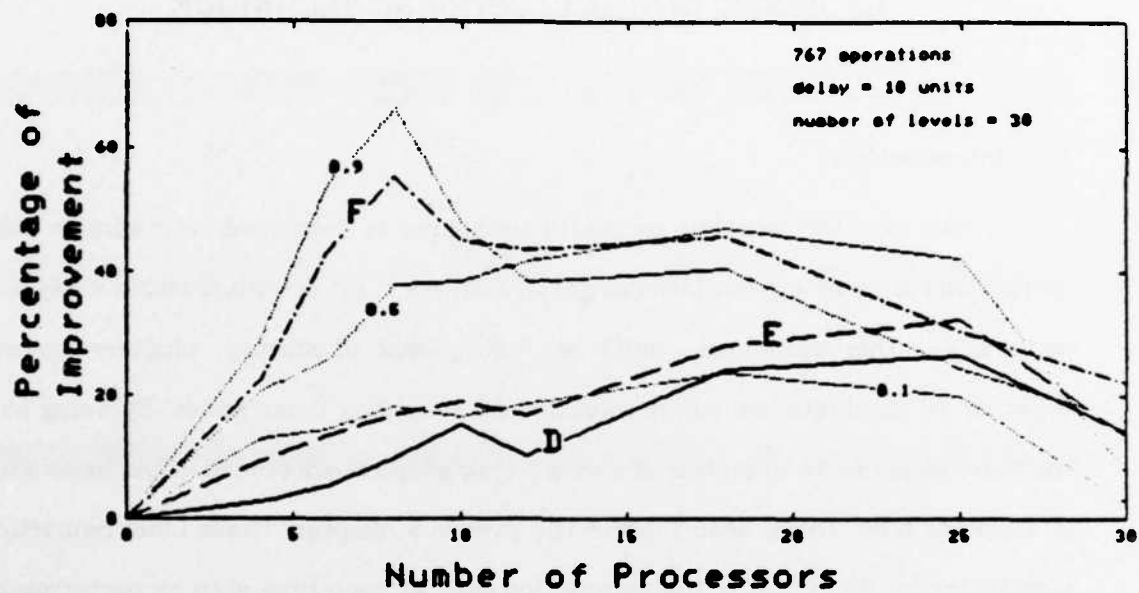
(a)



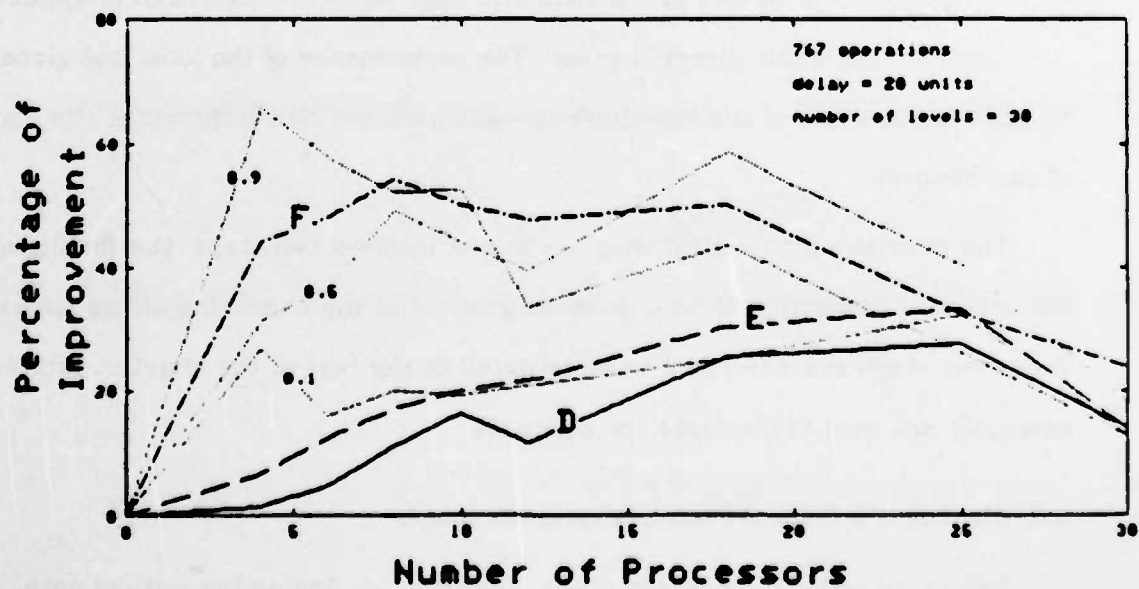
(b)

Figure 4.28 Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 767 Nodes (a) Delay of 1 Unit. (b) Delay Of 5 Units.





(a)



(b)

Figure 4.29

Comparison of The Speedup Performance Of Heuristics D, E, F, EF ( $\beta = 0.1, 0.5, 0.9$ ) For Task Graph With 767 Nodes (a) Delay of 10 Unit. (b) Delay Of 20 Units.

## CHAPTER 5

### HEURISTIC GLOBAL CLUSTERING TECHNIQUE

#### 5.1. Introduction

In this chapter, another heuristic technique is described to minimize the completion time of a given LU task graph with constant communication delay on each edge. This technique, which we call *global clustering*, clusters nodes together to eliminate the communication delay among these nodes. By doing so, the time required to complete the whole task graph is shortened. This heuristic is different from those described in the previous chapter. Those local heuristic algorithms try to minimize the completion time at each time step by performing combinatorial mapping between a set of ready tasks at that particular time step and the processors. In this global heuristic approach, the algorithm is applied iteratively on the whole directed graph. The performance of the local and global algorithms in terms of the speedup ratio achieved will be compared at the end of this chapter.

The heuristic global clustering technique involves two steps, the finding of the critical ( longest) path in a directed graph and the clustering of the nodes. These two steps are described in more detail in the rest of the chapter. Simple examples are used to illustrate the concepts.

#### 5.2. Finding the Critical Path in a Directed Graph

Before we go into the details of the algorithm for finding the critical path, a few definitions are necessary.

A *directed graph*  $G = (N, E, <)$  is a graph which consists of a set of nodes and a set of edges. Each node has a number for its identification and each edge has a

non-negative number  $w_{ij}$  associated with it, which we call its weight. It represents the communication delay between the two nodes connected by this edge. Each node also has its own weight  $\tau_i$  representing the node execution time. The precedence constraint between a pair of nodes is denoted by  $i < j$  which means that node  $i$  is the immediate predecessor of node  $j$ .

The *length* of a path is defined as the sum of the weights of the edges as well as the weights of the nodes which constitute that path.

The *critical path* is the length of the longest path from the node to the terminal node.

The algorithm for finding the critical path in a directed graph is called the Modified Cascade Algorithm. The idea comes from the original Cascade Algorithm for finding the shortest path in a directed graph. A few modifications are made to the original Cascade Algorithm in order to search for the longest path. Hence a careful understanding of the methods for finding the shortest path is necessary, of which there are quite a few. In most of the methods, a matrix called the distance matrix  $D$  associated with the graph is formed. The elements of the distance matrix  $d_{ij}$  are defined as

$$d_{ij} = \begin{cases} w_{ij} & \text{if } i < j, i \neq j; \\ 0 & \text{if } i = j; \\ \infty & \text{otherwise;} \end{cases}$$

In the case of an undirected graph, the distance matrix is symmetric while for a directed acyclic graph, the distance matrix is asymmetric.

For completeness, we will mention another method available for finding the shortest path in which the distance matrix is not necessary. It is based on the solution of a system of nonlinear equations obtained from the theory of dynamic programming[26].

### 5.2.1. Dynamic Programming Approach

Given a graph with  $N$  nodes, a system of equations in which  $f_i$  are the unknown quantities are formed

$$\begin{aligned} f_i &= \min_{j \in \mathcal{A}_i} (d_{ij} + f_j) \quad i = 1, 2, \dots, N-1 \\ f_N &= 0 \end{aligned} \quad (5.1)$$

where  $f_i$  = shortest distance from the node  $i$  to the destination node  $N$ .

The minimization is taken over all  $j$  which can be reached directly from  $i$ . The method of successive approximations is used to solve the above system according to the following steps:

- (1) Pick an initial guess  $f_i^{(1)}$  ( $i=1, 2, \dots, N-1$ ) and choose  $f_N^{(1)} = 0$ .
- (2) Update the next approximation  $f_i^{(k)}$  using the recursive relations:

$$\begin{aligned} f_i^{(k)} &= \min_{j \in \mathcal{A}_i} (d_{ij} + f_j^{(k-1)}) \\ f_N^{(k)} &= 0, \quad k = 2, 3, \dots \end{aligned}$$

After solving all the  $f_i$ 's, it is not difficult to deduce the shortest path. It has been proved in [1] that the method of successive approximations converges to a unique solution.

### 5.2.2. The Cascade Algorithm

This algorithm is based on the method by Farbey, Land and Murchland[27]. A brief description of the algorithm will be given, followed by a simple example. In the next section, the Modified Cascade Algorithm for finding the critical path in the LU task graph based on the modifications of the original Cascade Algorithm will be discussed.

First of all, let us define an elementary multiplication of two matrices, which is different from the ordinary matrix multiplication. The product  $C$  of two matrices  $A$  and  $B$  is defined as

$$c_{ij} = \min_k (a_{ik} + b_{kj}) \quad k = 1, 2, \dots, N \quad (5.2)$$

The Cascade Algorithm is essentially two successive squarings of the original distance matrix associated with the graph, with the multiplication of two matrices defined as above. Two things have to be kept in mind when carrying out this algorithm:

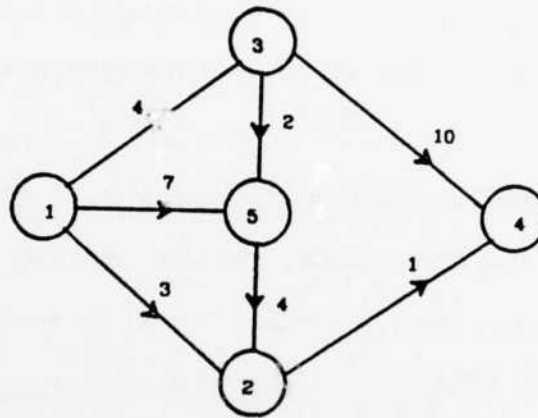
- (1) Elements in the matrix must be updated immediately as soon as they are calculated.
- (2) Elements must be calculated in the order  $d_{11}, \dots, d_{1N}; d_{21}, \dots, d_{2N}; \dots, d_{N1}, \dots, d_{NN}$ ; on the first squaring (forward process) and in the order  $d_{NN}, \dots, d_{N1}; d_{N-1N}, \dots, d_{N-1,1}; \dots, d_{1N}, \dots, d_{11}$  on the second squaring (backward process).

The resultant matrix  $S$  is the shortest distance matrix between any pair of nodes in the graph.

In many applications, the shortest path itself is required, i.e. the intermediate nodes which constitute the shortest path in going from node  $i$  to node  $j$  have to be known. This will be easily available if we make suitable records of the nodes during the course of the shortest distance calculation. For the purpose of keeping track of the shortest path, a *routing* matrix  $R$  is used. The  $r_{ij}$  element of  $R$  contains the number of the first node on a shortest path from node  $i$  to node  $j$ . The elements in  $R$  are changed as the algorithm proceeds. Initially, the  $r_{ij}$  is set equal to  $j$  for all  $i$  and  $j$ . During the shortest distance calculation, whenever we have to change an element  $d_{ij}$  in  $D$ , the corresponding element  $r_{ij}$  has to be updated also, i.e. whenever  $d_{ij} \leftarrow d_{ik} + d_{kj}$ ,  $r_{ij} \leftarrow r_{ik}$ .

If we are given a fully connected undirected graph of  $N$  nodes, we will have an  $N$  by  $N$  full distance matrix. If we call the addition of two elements in  $D$  and the comparison to obtain the minimum as one operation, the Cascade algorithm requires  $2N^3$  such operations. Hence it is a polynomial time algorithm.

The following example will illustrate how the Cascade algorithm works. Suppose a directed graph and the weights of the edges are given in the following figure :



The original distance matrix  $D$  and the routing matrix  $R$  are

$$D = \begin{bmatrix} 0 & 3 & 4 & \infty & 7 \\ \infty & 0 & \infty & 1 & \infty \\ \infty & \infty & 0 & 10 & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & \infty & \infty & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The matrix  $D$  and the matrix  $R$  after one step of the Cascade Algorithm are

$$D = \begin{bmatrix} 0 & 3 & 4 & 4 & 6 \\ \infty & 0 & \infty & 1 & \infty \\ \infty & \infty & 0 & 10 & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & \infty & \infty & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 2 & 3 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The final matrix  $S$  and the matrix  $R$  are

$$S = \begin{bmatrix} 0 & 3 & 4 & 4 & 8 \\ \infty & 0 & \infty & 1 & \infty \\ \infty & 8 & 0 & 7 & 2 \\ \infty & \infty & \infty & 0 & \infty \\ \infty & 4 & \infty & 5 & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 2 & 3 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 3 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 2 & 5 \end{bmatrix}$$

It can easily be verified for the above example that the entries in the final distances matrix  $S$  give the shortest distance between a pair of nodes in the directed graph and the routing matrix  $R$  gives the shortest path. For example, the shortest distance from node 3 to node 4 is 7 and the shortest path is from node 3 to node 5 ( $\tau_{34}$ ) to node 2 ( $\tau_{54}$ ) to node 4.

### 5.3. Modification Of The Cascade Algorithm To Find Critical Paths

There are basically two changes which have to be made to the Cascade algorithm in order to find the longest distance between any pair of nodes. The first modification which is quite obvious is in calculating the square of the distance matrix. In the definition of the product  $C$  of two matrices  $A$  and  $B$ , we change the minimum to maximum

$$c_{ij} = \max_k (a_{ik} + b_{kj}) \quad k = 1, 2, \dots, N \quad (5.3)$$

The second modification to the algorithm is on the entries of the distance matrix. Instead of assigning the entry  $d_{ij}$  the value of infinity ( $\infty$ ) if there is no edge from node  $i$  to node  $j$ , we just leave that entry empty. During the forward and backward processes, we will not consider that particular entry. The update procedure for the routing matrix stays the same.

The complexity of the modified algorithm is the same as the unmodified algorithm for a fully connected undirected graph. However, in most of the real applications, the graph is not fully connected and in some cases, it is directed. This implies that the distance matrix  $D$  is very sparse. Hence for efficient manipulation of matrix  $D$  in the Cascade algorithm and for efficient storage of a graph with a very large number of nodes, a special data structure for the distance matrix and the routing matrix is necessary. That will be the subject of next section.



### 5.3.1. Data Structure For Matrices D and R

The basic operation in the Cascade algorithm is the maximization of the sums of the elements in a certain row and the corresponding elements in the column in D. Hence nonzero entries in a row and a column should be linked together. A sensible choice is a uni-directional linked list across the rows and down the columns. With this simple data structure, only nontrivial operations are performed and the storage requirement will be much less even for a graph with a very large number of nodes. A similar data structure can be used for the routing matrix R. However we only link up those elements whose values have been updated. The other elements will have the same value as the corresponding column index.

We use the previous example to show how it works:

First the original distance matrix D and the routing matrix R are

$$D = \begin{bmatrix} 0 & 3 & 4 & & 7 \\ & 0 & & 1 & \\ & & 0 & 10 & 2 \\ & & & 0 & \\ 4 & & & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The matrix D and the matrix R after one step of the modified Cascade algorithm are

$$D = \begin{bmatrix} 0 & 11 & 4 & 14 & 7 \\ & 0 & & 1 & \\ & & 0 & 10 & 2 \\ & & & 0 & \\ 4 & & & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 5 & 3 & 3 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The final matrix S and the matrix R are

$$S = \begin{bmatrix} 0 & 11 & 4 & 14 & 7 \\ & 0 & & 1 & \\ & 6 & 0 & 10 & 2 \\ & & & 0 & \\ 4 & & 5 & 0 & \end{bmatrix} \quad R = \begin{bmatrix} 1 & 5 & 3 & 3 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 2 & 5 \end{bmatrix}$$

From the above S matrix, the length of the critical path is 14, and it goes from node 1 to node 3 ( $r_{14}$ ) to node 4.

#### 5.4. Application of Modified Cascade Algorithm to LU Task Graph

The above modified Cascade Algorithm cannot be applied directly to the LU task graph to find the critical path. The reason is that this algorithm finds the critical path in a directed graph without taking into account the weights on the nodes along the critical path. In other words, the algorithm chooses the longest path in a graph whose nodes have zero weights (the edges are still weighted). Recall that the nodes in the LU graph have weights which are the execution times. Hence some modifications to the distance matrix are necessary in order to use this Cascade algorithm. The simple idea of just adding the execution times of the starting and terminating nodes to the weight of the edge connecting them will not work. If we examine the algorithm carefully, that modification to the distance matrix might result in counting the execution times of those intermediate nodes lying on the path more than once. So that leads us to the idea of just adding the execution time of the starting node to the weight of the edge, and then perform the forward and backward processes of the Cascade algorithm on the modified distance matrix and update the routing matrix as usual. The distance matrix obtained will give the longest distance between any pair of nodes *without* taking into account the execution times of the terminating nodes of the edges. Hence we have to add to each nonzero entry in the distance matrix the corresponding execution time of the terminating node. Based on the nonzero entries in the final distance matrix, we can find out the critical path in the graph.

Perhaps it is a good idea to use our previous example to illustrate the above explanation. Let us use the notation  $\tau_i$  to denote the execution time of node  $i$  and suppose we have the following execution times for the nodes in the graph:

$$\tau_1 = 2;$$

$$\tau_2 = 1;$$

$$\tau_3 = 10;$$

$$\tau_4 = 1;$$

$$\tau_5 = 3;$$

The original distance matrix  $D$  and the routing matrix  $R$  are

$$D = \begin{bmatrix} 0 & 3 & 4 & & 7 \\ & 0 & & 1 & \\ & & 0 & 10 & 2 \\ & & & 0 & \\ 4 & & & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The distance matrix  $D$  after the addition of execution times of the starting nodes to the nonzero entries and the routing matrix  $R$  are

$$D = \begin{bmatrix} 0 & 5 & 8 & & 9 \\ & 0 & & 2 & \\ & & 0 & 20 & 12 \\ & & & 0 & \\ 7 & & & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

The distance matrix  $D$  after the forward process and the routing matrix  $R$  are

$$D = \begin{bmatrix} 0 & 18 & 8 & 28 & 18 \\ & 0 & & 2 & \\ & 19 & 0 & 21 & 12 \\ & & & 0 & \\ 7 & & 9 & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 5 & 3 & 3 & 3 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 3 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 2 & 5 \end{bmatrix}$$

The distance matrix  $D$  after the backward process and the routing matrix  $R$  are

$$D = \begin{bmatrix} 0 & 25 & 8 & 27 & 18 \\ & 0 & & 2 & \\ & 19 & 0 & 21 & 12 \\ & & & 0 & \\ 7 & & 9 & & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 3 & 3 & 3 & 3 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 3 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 2 & 5 \end{bmatrix}$$

The final distance matrix  $S$  after the addition of execution times of the terminat-

ing nodes to the nonzero entries and the routing matrix  $R$  are

$$S = \begin{bmatrix} 0 & 26 & 16 & 28 & 21 \\ & 0 & & 3 & \\ & 20 & 0 & 22 & 15 \\ & & & 0 & \\ & 8 & & 10 & 0 \end{bmatrix} \quad R = \begin{bmatrix} 1 & 3 & 3 & 3 & 3 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 5 & 3 & 5 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 2 & 5 \end{bmatrix}$$

From the final distance matrix  $S$ , the length of the critical path is 28. From the routing matrix  $R$ , the critical path starts at node 1 to node 3 ( $r_{14}$ ) to node 5 ( $r_{34}$ ) to node 2 ( $r_{54}$ ) to node 4 ( $r_{24}$ ).

### 5.5. Clustering Technique

Clustering refers to the grouping of nodes together, and nodes clustered together will be assigned to the same processor so that the communication delay among the nodes is eliminated. Which nodes should be clustered together is of primary importance. Recall that the main objective is the reduction of the completion time of the LU task graph. Hence the effect of clustering nodes together should accomplish this end. As pointed out in the earlier chapters, the length of the critical path has a close correlation with the completion time of the graph. Actually the length of the critical path without adding the delay along the path is the minimum time required to finish that task graph no matter how many processors are available. This is the best we can achieve. In the worst case assumption where each edge has a delay associated with it, the sensible approach is the shortening of the critical path by clustering nodes on the critical path.

#### 5.5.1. A Modified Heuristic Approach

This problem of reducing the communication overhead has been addressed by Jen[28] and Efe[29]. In this heuristic technique, a few modifications are made to their heuristics for arbitrary task graphs in order to obtain a more optimistic

approach. The definitions pertaining to the description of the heuristic are given below :

Let  $G = (N, E, <)$  be the original directed graph with  $N$  nodes and a set of edges  $E$ .

- (1)  $(i, j)$  = a directed edge from node  $i$  to node  $j$ .
- (2)  $t(i)$  = execution time for node  $i$ .
- (3)  $d(i, j)$  = communication delay from node  $i$  to node  $j$ .

Let  $N_c$  be a set of nodes clustered into a big node  $C$ . The reduced graph  $G_r = (N_r, E_r, <)$  of  $G$  is defined as

- (1)  $N_r = N - N_c + C$  ;
- (2) For all  $i$  in  $N_c$ ,  $j$  in  $N - N_c$  :  
 $(c, j) = (i, j) ; (j, c) = (j, i) ;$
- (3)  $t(c) = \sum t(i)$  for all  $i$  in  $C$  ;
- (4)  $d(c, j) = \text{Max } d(j, i)$  for all  $i$  in  $C$  ;
- (5)  $d(j, c) = \text{Max } d(j, i)$  for all  $i$  in  $C$  ;

Consider an example. Figure 5.1(a) is the given precedence graph with six nodes. The communication delay between a pair of nodes is as shown. Suppose we decide to cluster nodes 2, 3 and 4 together in order to eliminate the delay between these nodes. The resultant reduced graph is shown in Figure 5.1(b).

The reduced graph has four nodes and the clustered node  $C$  consists of nodes 2, 3 and 4 of the original graph. The delays  $d_{23}$  and  $d_{34}$  are set to zero and the other delays are shown in Figure 5.1(b). The execution times of the clustered node  $C$  is equal to the sum of the execution times of the nodes 2, 3 and 4. We assume that the nodes in the clustered nodes are executed sequentially by one processor. However a further assumption in [28] is that data needed by other nodes are *not* transmitted until the end of the execution of all the nodes in that

cluster. This will certainly delay the completion time of the task graph. In the previous example, nodes 2, 3 and 4 are executed before the transmission of the data to nodes 5 and 6. In our heuristic approach, once a node in the cluster is executed and if the result is needed by some other nodes, the result is transmitted *immediately*. In the example, results from nodes 2 and 3 will be sent to node 5 without waiting for node 4 to finish its execution. Hence the execution of node 5 will not be delayed.

Another feature of the heuristic is the load-balancing constraint. The load-balancing constraint limits the maximum number of nodes a cluster can have. This reduces the possibility of overloading the processors when scheduling is performed on the reduced task graph. The reason to implement the load-balancing constraint is that it is not uncommon to find that a large number of nodes are clustered together in some clusters while there may be relatively fewer number of nodes in some other clusters. This introduces the effect of uneven loading on the processors. In other words, some processors may be assigned to execute a large number of nodes compared to other processors. This results in the effect of unbalanced allocation of resources and it might also produce an undesirable schedule when a scheduling algorithm is applied to the reduced graph. The load-balancing constraint has the disadvantage of not producing the shortest critical path in the heuristic sense. The reason is that nodes that must be clustered together in order to give a shorter critical path are forced to break apart to form more clusters because of the constraint on the maximum number of nodes a cluster can have. Hence the communication delay will remain between these clusters thus lengthening the critical path.

Simulation programs were written and run on several examples and the results are summarized in the next few sections. A structured and high level description of the simulation programs is given in the following section.

### 5.6. High Level Description of the Heuristic Clustering Technique

After describing the finding of the critical path and the clustering technique, we are ready to give a more detail description of the heuristic global clustering algorithm. Let us adopt the following notations:

$G$  : precedence graph

$L$  : critical path of  $G$

$G'$  : new precedence graph

$L'$  : critical path of  $G'$

$N_c(i)$  : number of nodes in cluster  $i$

$\text{Max\_nd}$  : Maximum number of nodes a cluster can have

#### ALGORITHM

Initialization :

Given an initial precedence graph,  $G$ . Specify  $\text{Max\_nd}$  and initialize the length of  $L' = \infty$ .

for( ; ; ) /\* forever loop \*/

(1) Find the critical path  $L$  in  $G$ .

If ( the length of  $L' < \text{length of } L$  )      exit

Else      continue

(2) Unmark all the edges on  $L$ .

(3) Find node  $i$  and node  $j$  on  $L$  such that edge( $i, j$ ) is unmarked.

If ( no such edge exists )      exit

Else      continue

If ( neither node  $i$  nor node  $j$  is in a cluster )

create a new cluster containing these two nodes. Set the delay

$d(i, j)$  to zero.



Else if( either node  $i$  or node  $j$  is in an existing cluster  $k$  ),

$N_c(k) = N_c(k) + 1;$

{ If (  $N_c(k) > \text{Max\_nd}$  ),

mark edge( $i, j$ ), go to (3).

Else

group the node ( $i$  or  $j$ ) not in the cluster into the cluster  $k$   
and set all the delays between that node and the nodes in  
cluster  $k$  to zero. i.e.

$d(i, c) = 0;$  for all  $c$  in cluster  $k$  if node  $j$  is in cluster  $k$ .

$d(j, c) = 0;$  for all  $c$  in cluster  $k$  if node  $i$  is in cluster  $k$ .

} /\* end of Else if \*/

Else if ( node  $i$  is in cluster  $p$  and node  $j$  is in cluster  $q$  )

{ if (  $N_c(p) + N_c(q) > \text{Max\_nd}$  )

mark edge( $i, j$ ), go to (3)

else

group all the nodes in clusters  $p$  and  $q$  together and set all  
the delays between the nodes in clusters  $p$  and  $q$  to zero. i.e.

$d(i, c) = 0;$  for all  $c$  in cluster  $q$

$d(j, d) = 0;$  for all  $d$  in cluster  $p$

} /\* end of Else if \*/

(4) Call the reduced graph ( graph after clustering )  $G'$ , and find its  
critical path  $L'$ . Set  $G = G'$ , go to (1).

} /\* end of forever loop \*/

### 5.6.1. Data Structure for Managing the Clusters

Note that in this heuristic algorithm, keeping the records of which nodes are in which clusters is very important. This determines which delay on an edge will be set to zero. In this section, a data structure for manipulating these

clusters will be described. This data structure will have the flexibility of adding new clusters as they are formed, keeping track of what nodes belong to which cluster, inserting new nodes to the existing clusters and grouping clusters together. A uni-directional linked list is used for the above purpose. The structure has three fields defined as

```
struct node_cluster
{
    int node_no ;
    int cluster ;
    struct node_cluster *pt ;
}
```

The first field ( int node\_no ) identifies the node number of the node. The second field ( int cluster ) tells which cluster that particular node belongs to. The third field ( struct node\_cluster \*pt ) is a pointer pointing to the next structure of the same kind. An example is shown in Figure 5.2. Nodes 3 and 10 are in cluster 2 and nodes 12 and 14 are in cluster 3. Suppose in the clustering process, we find that node 3 and node 7 should be clustered together. By transversing through the existing linked list, we discover that node 3 is in cluster 2. Hence we add the new node 7 to the linked list as shown in Figure 5.2(b). In this example, cluster 2 contains nodes 3, 7 and 10. A new cluster which is not in the existing linked list can be inserted similarly. Note that nodes are inserted in such a way that they are arranged in ascending order according to their node numbers for easy searching. Once a pair of nodes for clustering is identified, the delays on those edges that need to be eliminated are found according to step (3) in the algorithm described in the previous section. We can go directly to the distance matrix associated with the graph and set the values of the appropriate entries to zero.

### 5.7. Complexity of the Heuristic Clustering Algorithm

The dominant cost in the heuristic clustering algorithm is the finding of the critical path in the directed graph. As mentioned in section 5.2.2, the Cascade Algorithm requires  $2N^3$  operations for a  $N$  by  $N$  full distance matrix ( corresponding to a fully connected directed graph with  $N$  nodes ) where one operation is defined as the addition of two elements in the distance matrix and the comparison to obtain the maximum. The other steps like accessing the data structures of the clusters, the distance matrix and the routing matrix take substantially less time compared to the Cascade Algorithm. For example finding whether a node is in one of the clusters requires at most  $N$  comparisons when transversing the linked list of the clusters. Since we have to repeat the finding of the critical path and accessing the necessary data structures iteratively until the shortest critical path is attained, the worst case takes  $N \times 2N^3$  operations by going through all the nodes in the graph. Hence the complexity of the heuristic is at most  $O(N^4)$ .

However in most of the applications, the directed graph is not fully connected. This is the case for the LU task graph, since in the Doolittle Algorithm there are no loops in the directed graph. Furthermore, the two operations, update and divide, at most have to receive three values in order to carry out the calculation. This implies that the in-degree ( number of edges going into a node ) of a node ( an operation ) is at most three. Hence most of the distance matrices associated with a LU task graph are extremely sparse. With the linked list data structure for the distance matrix, the complexity of the heuristic clustering algorithm is expected to be much less than  $O(N^4)$  where  $N$  is the number of nodes in the LU task graph.

## 5.8. Simulation Results

In this section, several directed graphs with different number of nodes are tested on this heuristic clustering algorithm. One graph is generated from the circuit matrix of a bench-mark circuit in[1]. The others are generated randomly. These examples are run on a VAX 11/780 computer. Three different simulation results were obtained. The most important one shows the effectiveness of reducing the length of the critical path of a given graph as a function of the constraint on the maximum number of nodes allowed in one cluster. The second simulation results shows the reduction of the number of nodes of the reduced graph as a function of the constraint on the maximum number of nodes in a cluster. The last simulation results compare the performance of this heuristic clustering technique with those local heuristic algorithms described in chapter four. The performance is measured based on the speedup ratio obtained on these examples under the assumption that enough processors are available to achieve the maximum speedup ratio. This speedup performance is plotted as a function of delay for all the heuristic algorithms. For the previous two simulations, delay is a parameter in the curves. Each of the above three simulations is described in more detail in the following sub-sections.

### 5.8.1. Reduction of the Length of the Critical Path

The performance is measured as the percentage of reduction on the original length of the critical path of a given directed graph. The percentage of reduction is defined as

$$\% \text{ of reduction} = \frac{\text{orig\_len}(\text{graph}) - \text{clust\_len}(\text{graph})}{\text{orig\_len}(\text{graph})} \times 100$$

where  $\text{orig\_len}(\text{graph})$  = Length of the original graph.  $\text{clust\_len}(\text{graph})$  = Length of the final reduced graph. Two sets of results are generated with delay as a parameter. The first set of results shows the situation where data needed by

other processors is transmitted once it is generated in a clustered node. The results are shown from Figure 5.3(a) to Figure 5.3(c). The second set of results assumes the case where all the nodes in a cluster are executed before transmitting the data needed by other processors. It is expected that the first case gives a better performance than the second case as we have already addressed the above problem in the section describing the heuristic algorithm. The second set is shown from Figure 5.4(a) to Figure 5.4(c). These show that as the number of nodes per cluster increases, the percentage of reduction increases until a certain point is reached where further clustering is not necessary. In all cases, as seen from the simulation results, the clustering technique becomes more effective as the delay in the graph increases.

#### 5.8.2. Reduction on the Number of Nodes

When the clustering technique is applied to a graph, nodes are grouped into clusters. By the definition of the reduced graph, nodes in a cluster are executed in sequence and the cluster is considered as one node during the next iteration of the algorithm. Hence the number of nodes in the reduced graph will have *fewer* nodes than the original graph. This will have a desirable effect on the computational time spent in the assignment of the nodes to processors. Recall that most of the feasible scheduling algorithms have polynomial running time in  $N$  where  $N$  is the number of nodes to be scheduled in a directed graph. Hence clustering nodes together will reduce the time in scheduling the nodes in the resultant reduced graph.

The performance of the heuristic in this sense is measured as the percentage reduction of the number of nodes. It is defined as

$$\% \text{ of reduction} = \frac{\text{orig\_num}(\text{graph}) - \text{clust\_num}(\text{graph})}{\text{orig\_num}(\text{graph})}$$

where  $\text{orig\_num}(\text{graph}) =$  number of nodes in the original graph.

$clust\_num(graph)$  = number of nodes in the reduced graph. The results are shown from Figure 5.5(a) to Figure 5.5(c) where the two curves correspond to two cases mentioned in the subsection 5.4.1. The curve B corresponds to the optimistic case where data is transmitted immediately once the node is executed in the cluster. The curve A corresponds to the pessimistic case where data is transmitted only after all the nodes in the cluster are executed. The same conclusion is observed as in the reduction of the length of the critical path. The percentage reduction increases as the constraint on the number of nodes in the cluster increases until it reaches a point where further clustering will not reduce the number of nodes in the graph.

The simulation results discussed in the previous two subsections reflect the performance of the global heuristic clustering technique measured on the reductions on the length of the critical path and the number of nodes on the directed graph. In the following subsection, this heuristic will be compared with the local heuristics already described in chapter four on how much speedup can be achieved when they are applied to schedule nodes on processors.

### 5.8.3. Comparison of Global and Local Heuristic Techniques

The global heuristic clustering technique presented in this chapter, in a certain sense, provides an alternative way of scheduling nodes to the processors in the presence of communication delay. The comparison of the global and local heuristic techniques on the performance is measured as the speedup ratio achieved under the assumption that enough processors are available to obtain the maximum speedup. Under this assumption, the completion time of the reduced graph is the length of the critical path on that graph. We also make the comparison under the optimistic global clustering case in which the result needed by other processors is transmitted immediately once it is computed in the cluster.

The speedup ratio is defined as

$$\text{speedup ratio} = \frac{\text{completion time using single processor}}{\text{completion time using } m \text{ processors}}$$

where  $m$  is the number of processors needed to achieve the maximum speedup.

The same four example graphs are used for the purpose of comparison of the performance as a function of communication delay. The results are shown from Figure 5.6(a) to Figure 5.6(d). They show that the global clustering technique attains better speedup ratio than the local heuristics, except in the example graph with 18 nodes, where it achieves the same speedup as the heuristic D, E and F ( see Figure 5.6(b) ). In general, the global technique gives in the range of 50% to 100% better speedup performance than the local heuristics.

## 5.9. Conclusion

This chapter presents the basic techniques used in the global clustering algorithm. It performs node clustering on the critical path of a directed graph to eliminate the communication delay. The iterated process of clustering is done on the whole graph as compared to the technique of minimizing the completion time at each time step described in the last chapter. Four example graphs are used to test out its performance measured on the reduction on the length of the critical path, the reduction on the number of nodes and the speedup ratio achieved compared to the local heuristic techniques. The simulation results show that the global clustering technique performs better than the four local heuristics.

A final remark is that the execution times of the nodes in the reduced graph are different after the clustering technique is applied. However, the heuristic scheduling techniques described in chapter four still can be employed to schedule the nodes of the reduced graph to the processors. Some modifications are necessary to the computer programs written for the local



heuristics to take into account the different execution times of the nodes when the elapsed times of the processors are calculated. A major addition is needed to the programs of the clustering technique so that the reduced graph is in the correct input format to the local heuristics described in the last chapter.

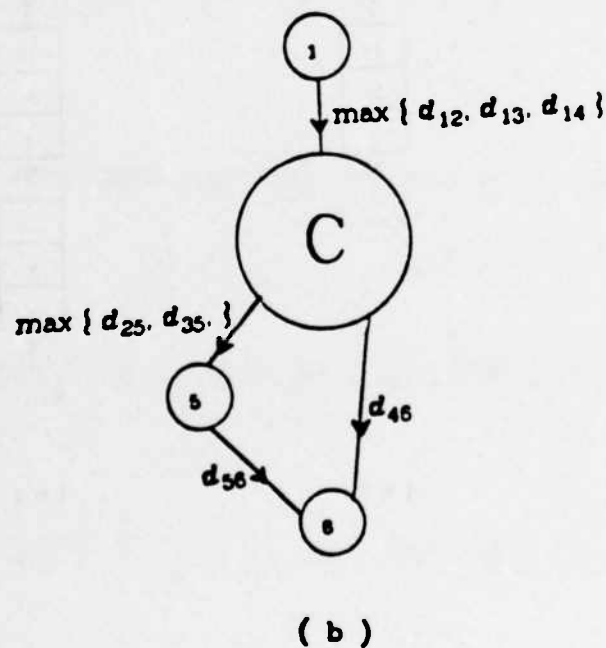
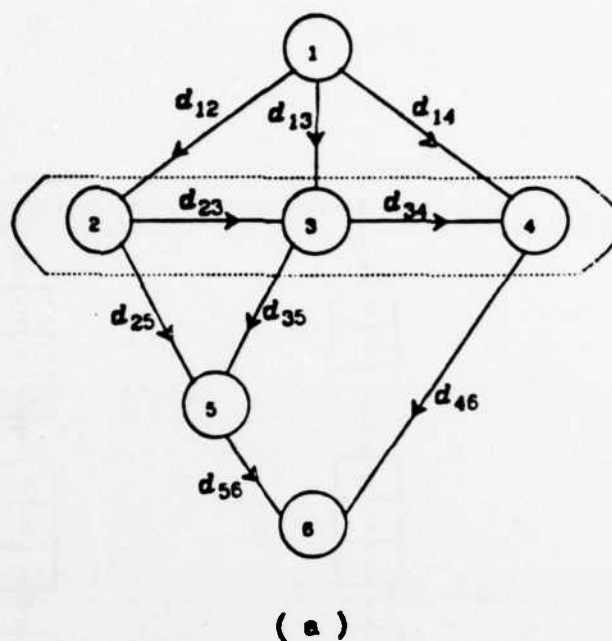


Figure 5.1 An Example Of Clustering Nodes In A Directed Graph. (a) The Original Graph (b) The Reduced Graph After Nodes 2, 3 and 4 Are Clustered.

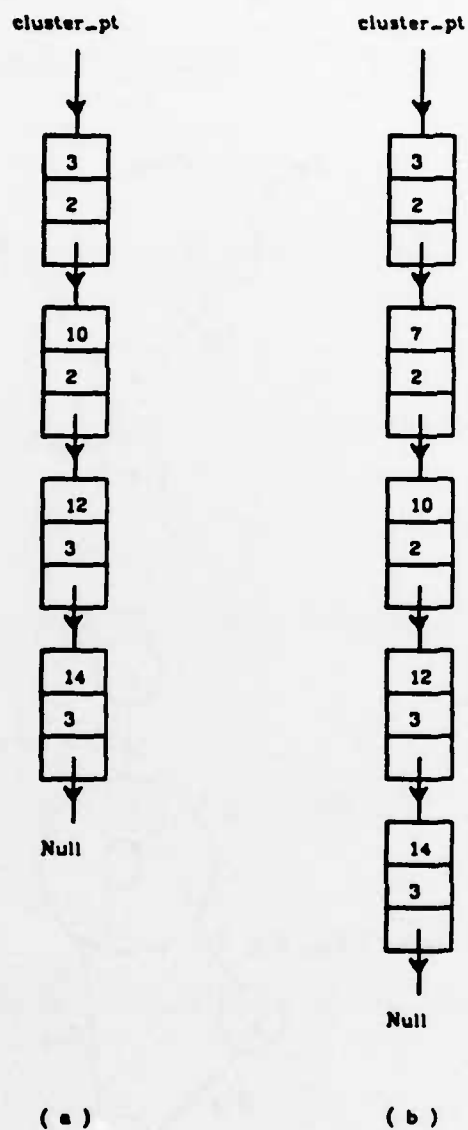


Figure 5.2

Data Structure For Clusters. (a) Data Structure For Clusters 2 And 3 (b) Data Structure After Node 7 Is Clustered Into Cluster 2.

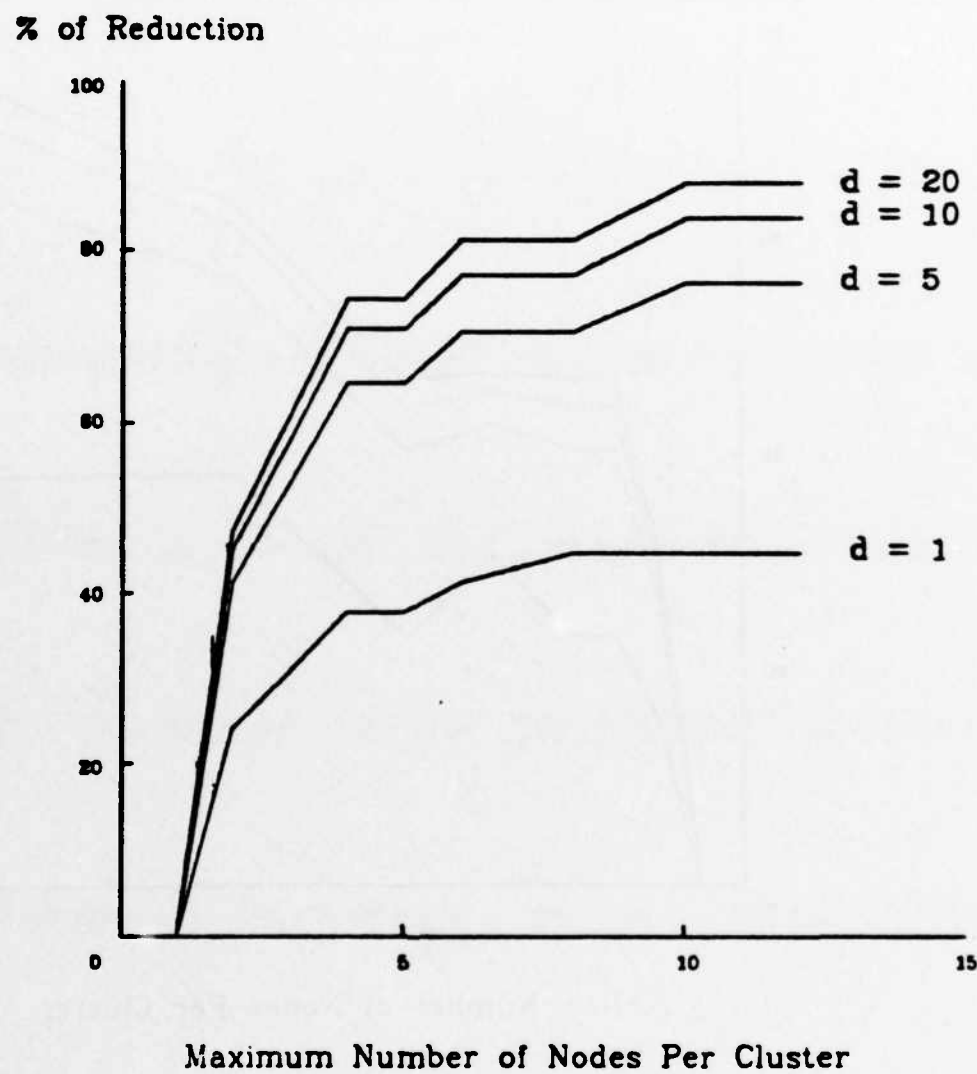


Figure 5.3(a) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of Each Node In A Cluster. Example Graph With 18 Nodes

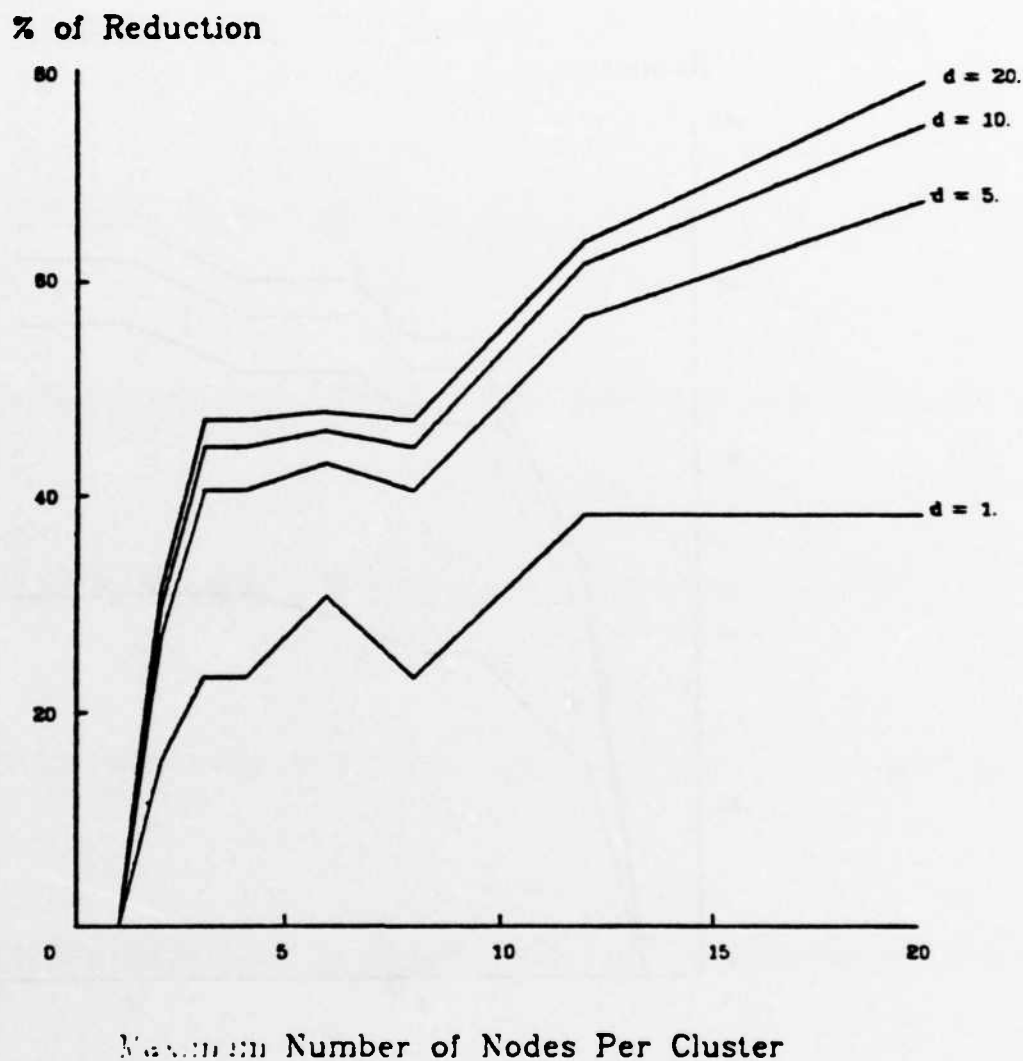


Figure 5.3(b) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of Each Node In A Cluster. Example Graph With 32 Nodes

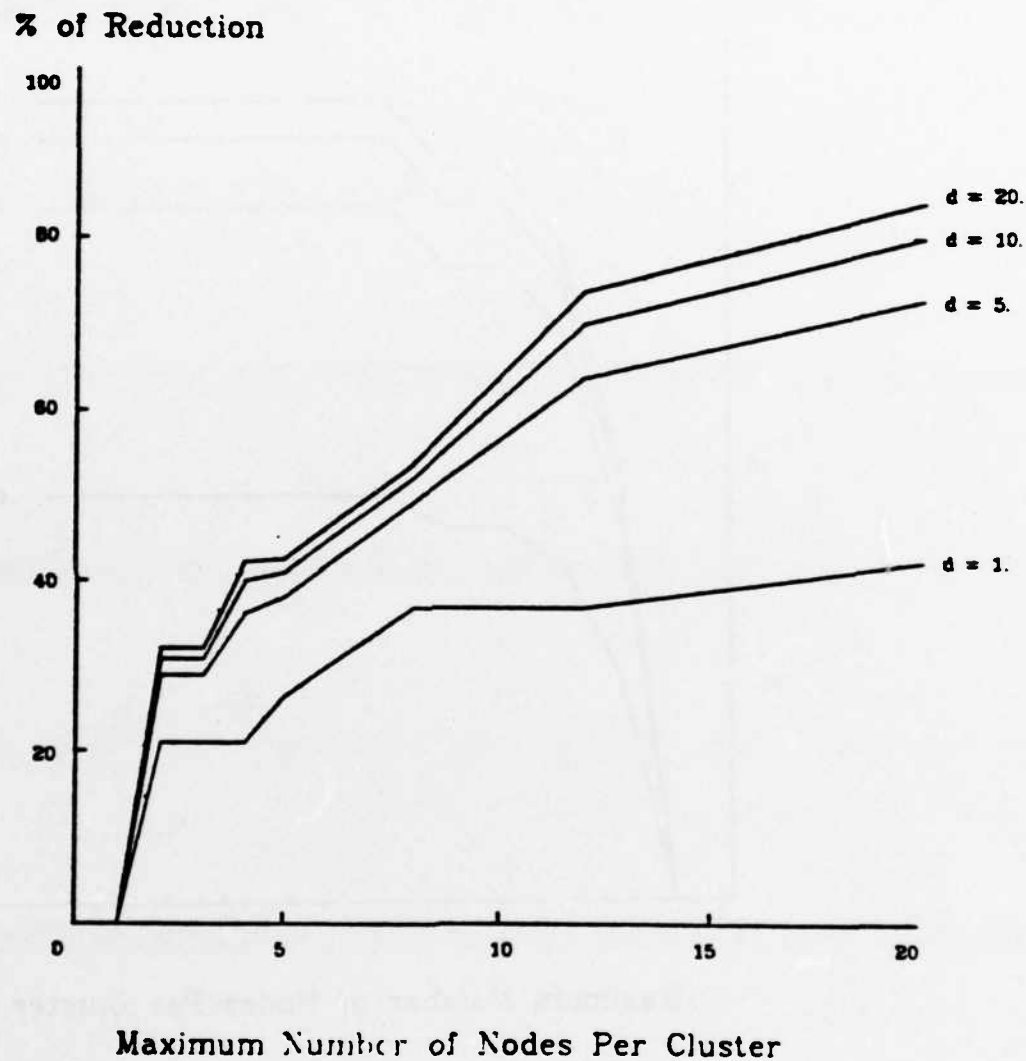


Figure 5.3(c) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of Each Node In A Cluster. Example Graph With 58 Nodes

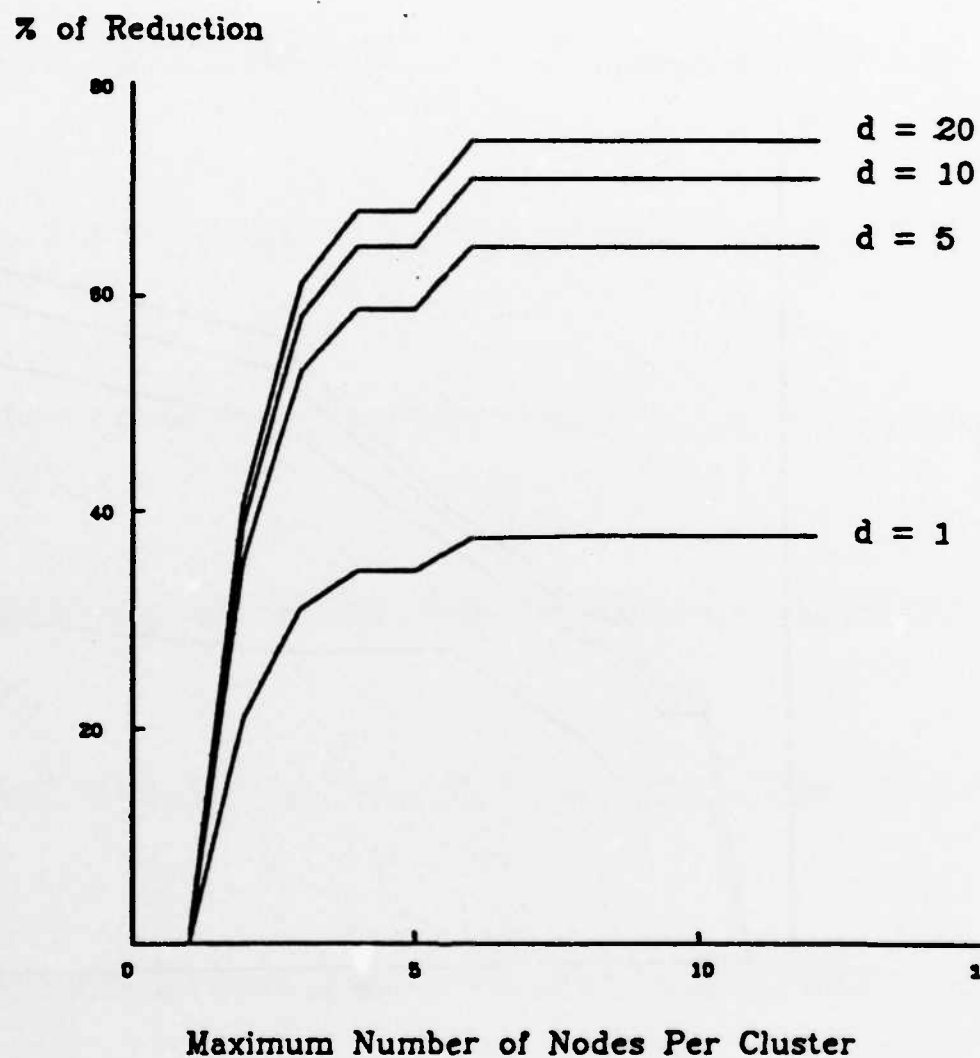


Figure 5.4(a) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of ALL The Nodes In A Cluster. Example Graph With 18 Nodes



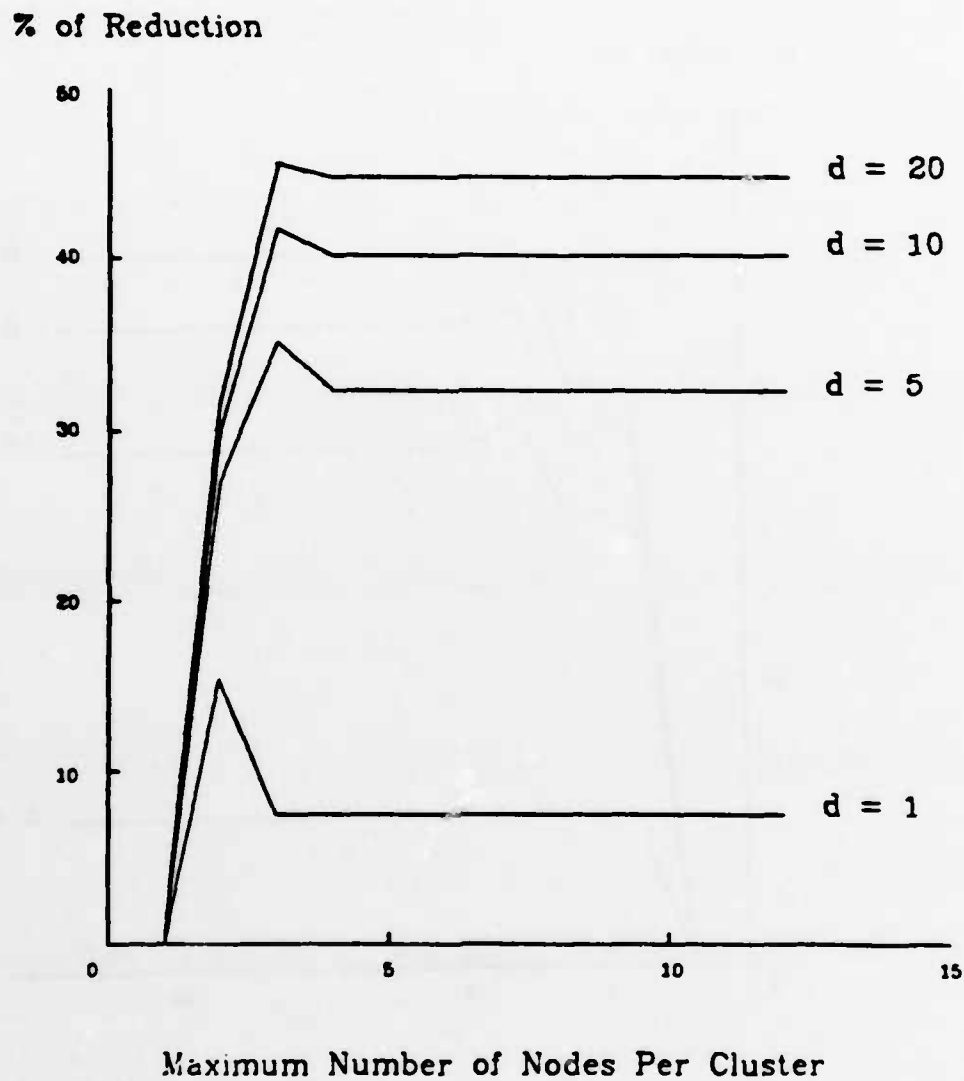


Figure 5.4(b) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of ALL The Nodes In A Cluster. Example Graph With 32 Nodes

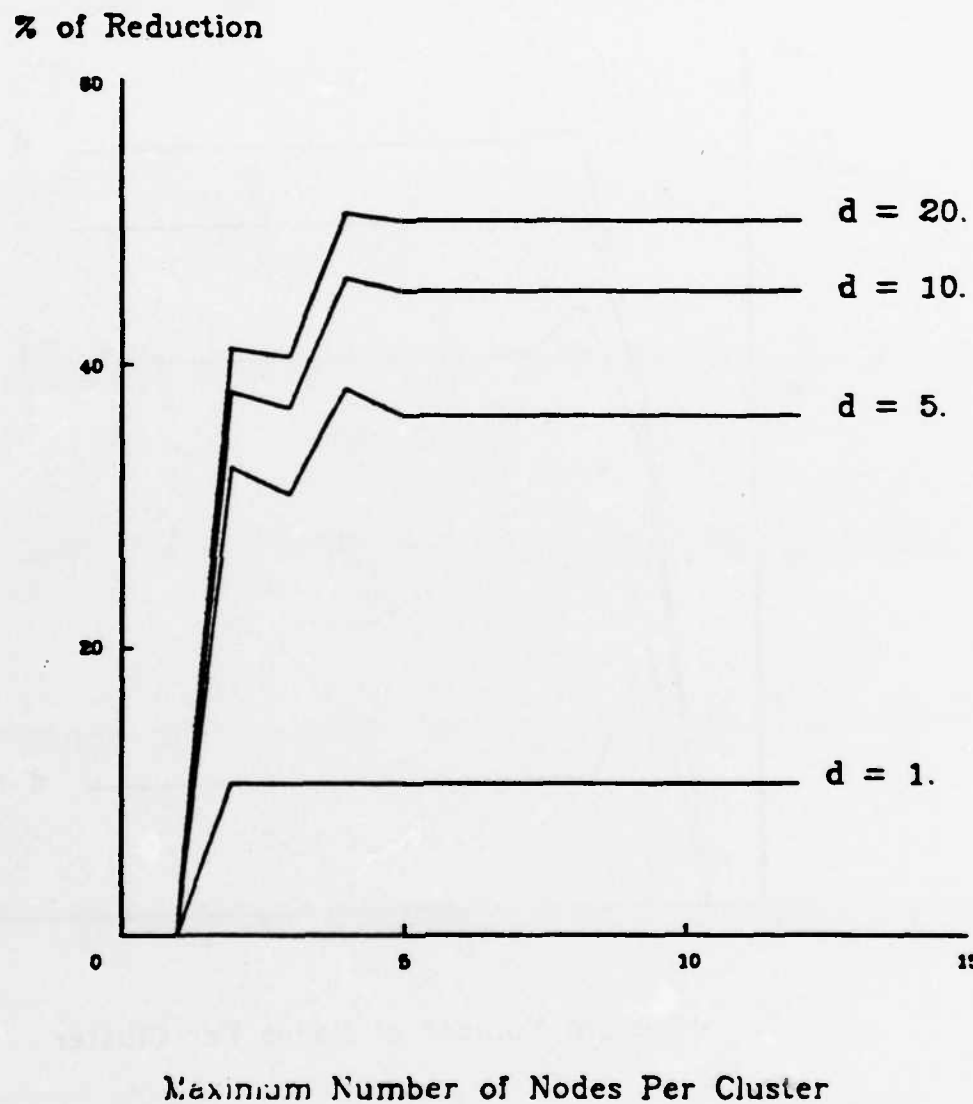


Figure 5.4(c) Reduction On The Length Of The Critical Path With Different Delays. Data Is Transmitted After The Execution Of ALL The Nodes In A Cluster. Example Graph With 58 Nodes

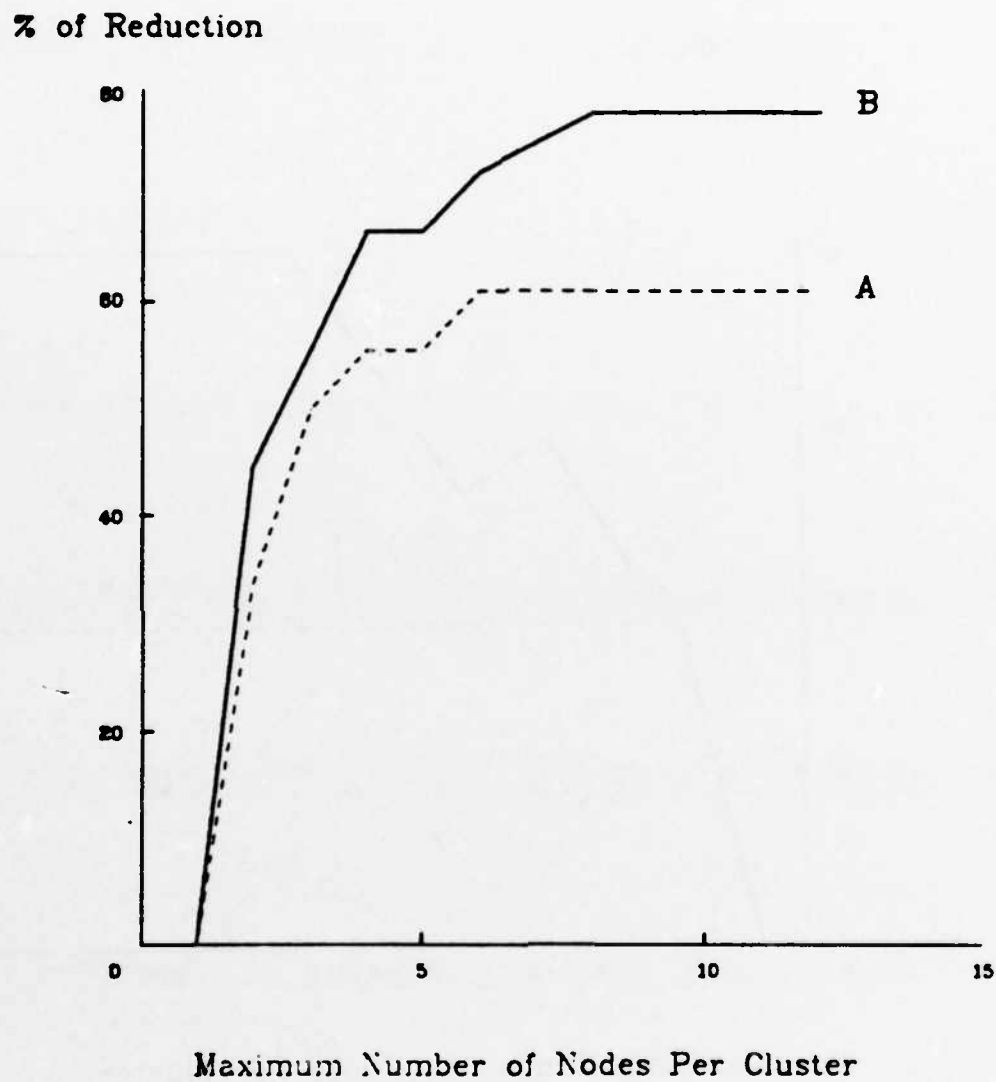


Figure 5.5(a) Reduction On The Number Of Nodes. Curve A Corresponds To The Pessimistic Case. Curve B Corresponds To The Optimistic Case. Example Graph With 18 Nodes

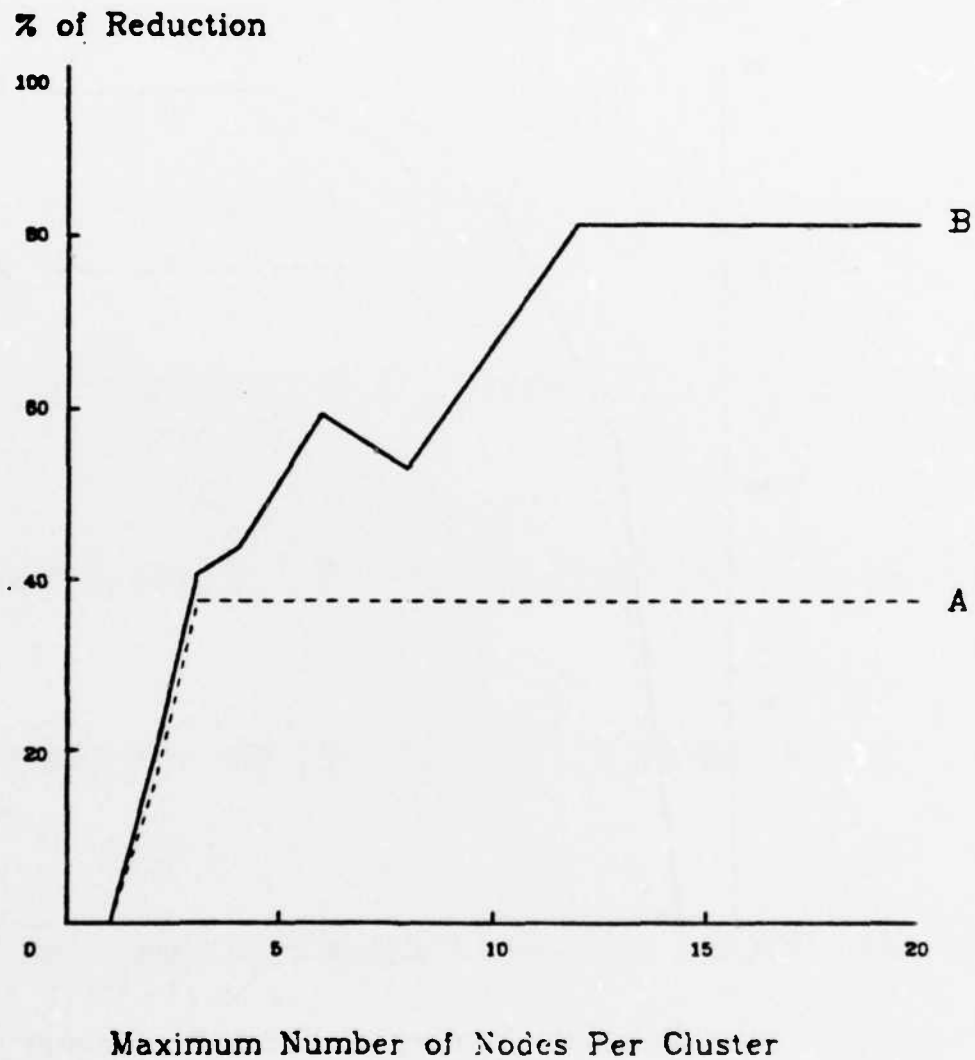


Figure 5.5(b) Reduction On The Number Of Nodes. Curve A Corresponds To The Pessimistic Case. Curve B Corresponds To The Optimistic Case. Example Graph With 32 Nodes

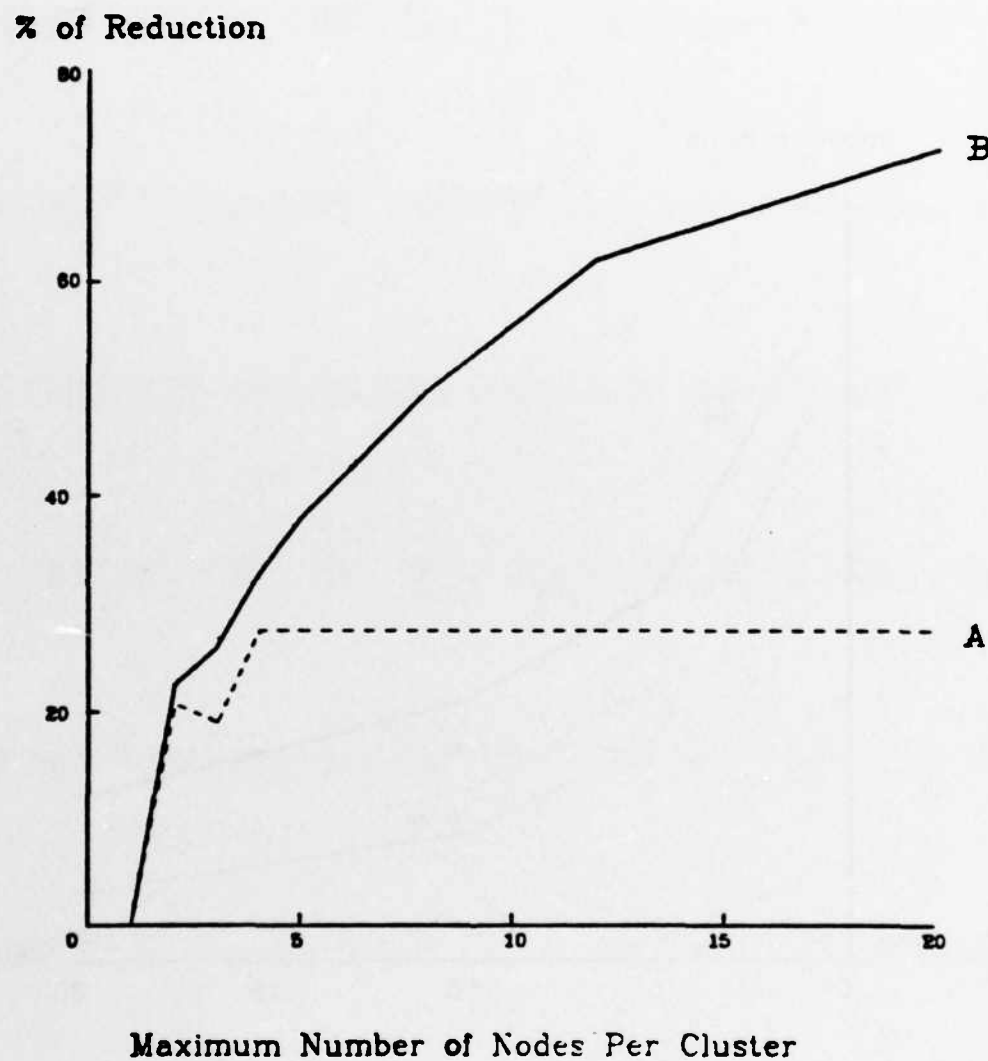


Figure 5.5(c) Reduction On The Number Of Nodes. Curve A Corresponds To The Pessimistic Case. Curve B Corresponds To The Optimistic Case. Example Graph With 58 Nodes

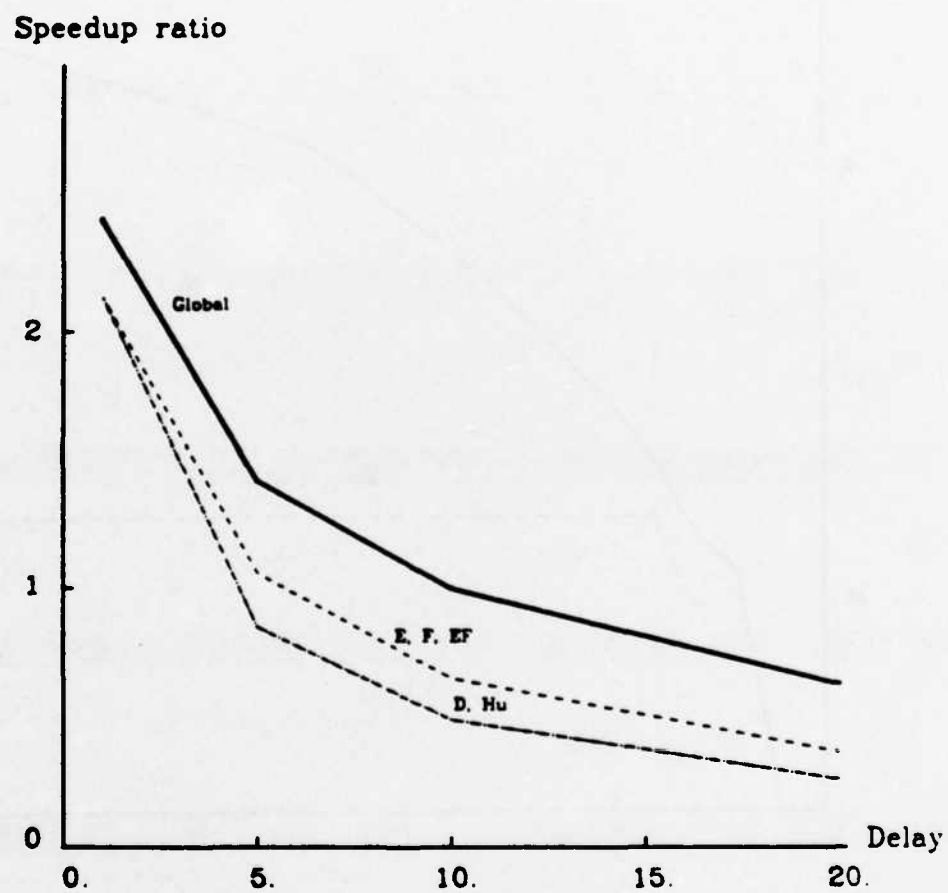


Figure 5.8(a) Comparison Of Global And Local Heuristic Techniques. Example Graph With 17 Nodes .

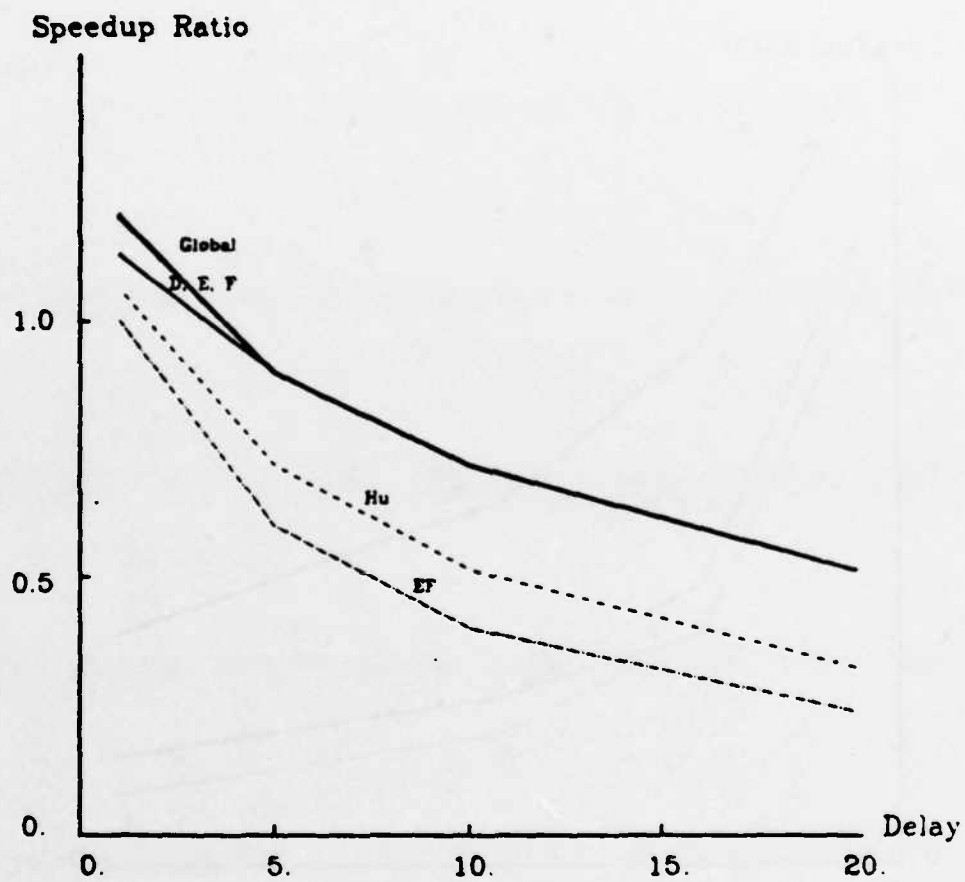


Figure 5.6(b) Comparison Of Global And Local Heuristic Techniques. Example Graph With 18 Nodes .



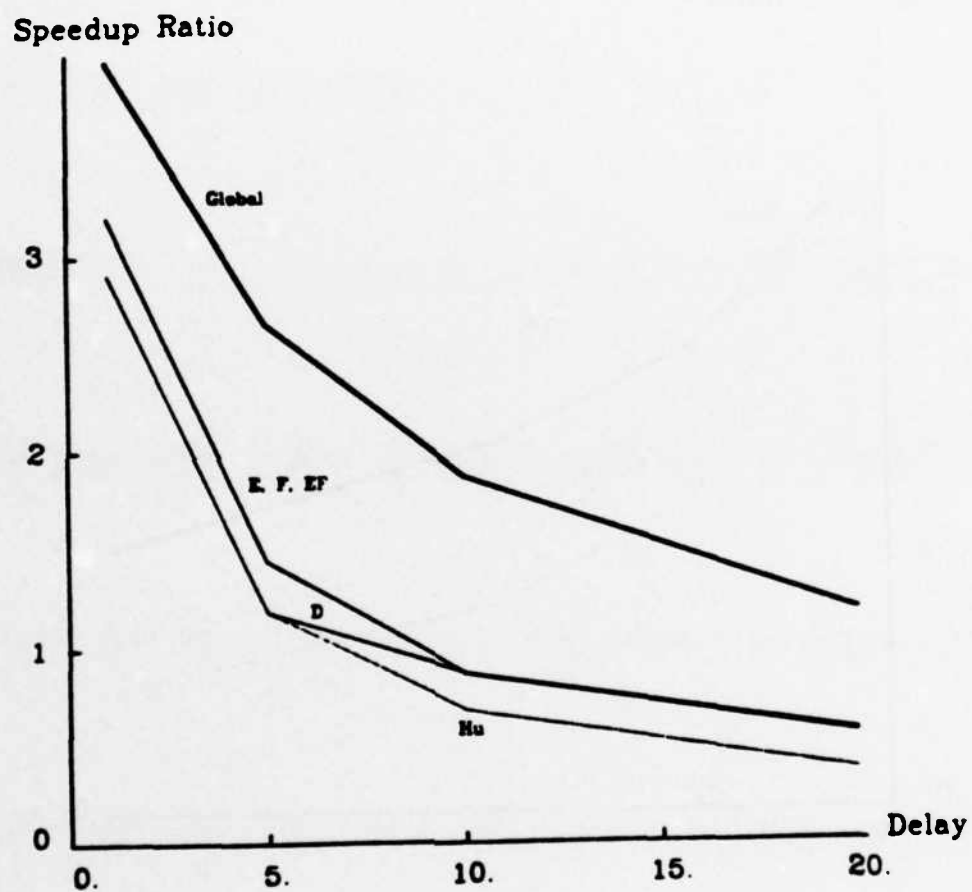


Figure 5.6(c) Comparison Of Global And Local Heuristic Techniques. Example Graph With 32 Nodes .

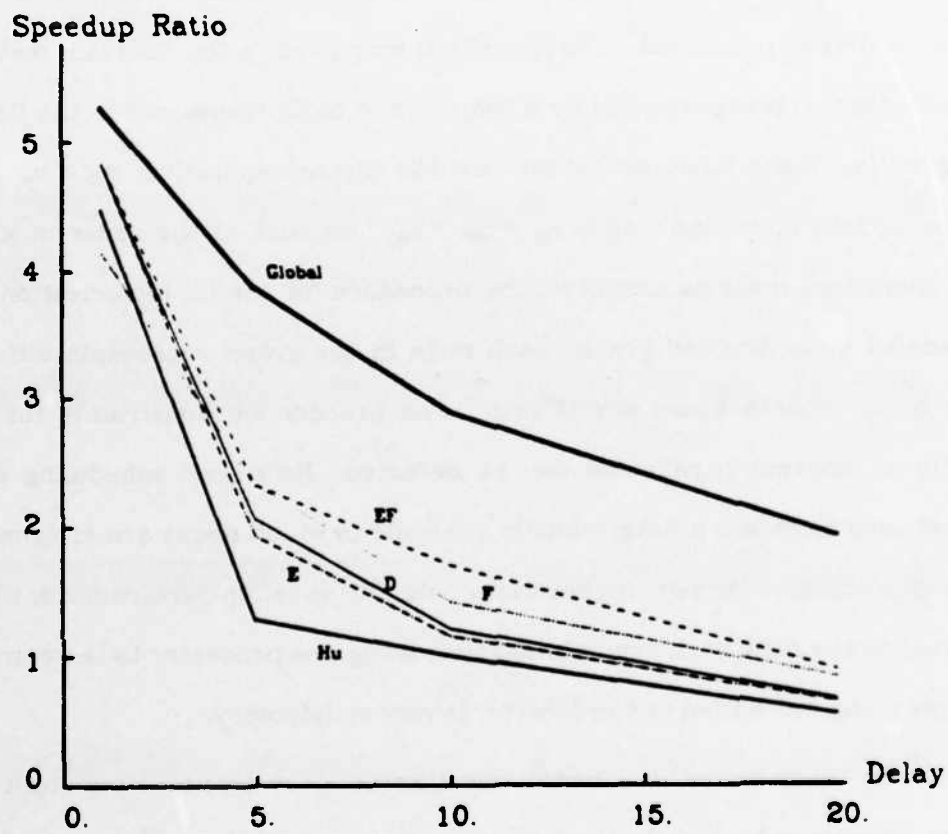


Figure 5.6(d) Comparison Of Global And Local Heuristic Techniques. Example Graph With 58 Nodes .

## CHAPTER 6

### CONCLUSION

#### 6.1. Overall Review of the Problem

In this dissertation the problem of decomposing a nonsingular, unstructured sparse matrix into a product of a lower triangular matrix and an upper triangular matrix is presented. The algorithm employed is the Doolittle method. The factorization is represented by a sequence of basic operations in the Doolittle algorithm. These basic operations are the divided operation:  $a_{ji} = a_{ji} / a_{ii}$  and the update operation:  $a_{kj} = a_{kj} - a_{ki} * a_{ij}$ . Because of the order in which these operations must be executed, the procedure for the LU factorization can be modeled by a directed graph. Each node in the graph represents either a divide or an update operation. Using these precedence constraints for the operations, inherent parallelism can be detected. Hu's level scheduling algorithm is used to obtain a deterministic schedule in which nodes are assigned to different processors for concurrent execution. The speedup performance, which is defined as the ratio of the completion time using one processor to the completion time using more than one processor, is very satisfactory.

However when there is a communication delay in transmitting from the sending processor to the destination processor, results have shown that the speedup performance based on the schedules obtained from scheduling algorithms without communication delay consideration degrades substantially. Hence other scheduling techniques with communication delay consideration should be developed. These scheduling problems are inherently intractable and heuristic scheduling algorithms provide a plausible approach.

In these heuristic techniques, combinatorial optimization algorithms such

as min\_max matching, weighted matching and heuristic clustering techniques are employed. In most of the cases, these heuristic scheduling techniques do produce schedules which have shorter completion time than the schedules obtained from Hu's level scheduling algorithm.

### **6.2. Significance of this Research**

The mapping of an algorithm to a directed graph model representation can be applied to many algorithms in other areas such as digital signal processing. The parallelism in the graph can be detected. The nodes in the graph do not necessarily represent one arithmetic expression, they can be segments of code or modules such as do loops in a computer program. In a realistic distributed processing environment, the interprocessor communication overhead is unavoidable. In many of the application algorithms, the graph model representation or the data dependency graph does not have special structure. In order to fully exploit parallel computing, scheduling methods should be constructed to minimize the effects of the communication overhead. This dissertation attempts to develop these techniques heuristically. Satisfactory results based on the simulation are obtained.

### **6.3. Future Related Research**

In this section, some of the possible future related research areas are discussed. Here the execution time for the nodes in the task graph have been assumed to be the same. However, the heuristic techniques described in chapters four and five can be extended to the case of unequal node execution times. In more practical situations, the nodes or in general the modules do not have equal processing times. Since the heuristic techniques developed also consider the node execution time when assigning node to processors, it is expected that promising results will be obtained.

When the interprocessor communication is ignored, bounds on the maximum and minimum number of processors required to finish the task graph in the shortest time can be obtained quite easily given the structure of the task graph. It is observed in the simulation results presented in chapter four that in the case where communication delay is taken into account, using more processors does not necessarily improve the speedup performance. In fact the increase in the number of processors reduces the speedup ratio compared to using small number of processors. This is perhaps due to the fact that large number of processors will increase the possibility of assigning nodes and their predecessors to different processors. Because of the delay between the communicating processors, it will increase the completion time of the task graph. In this case, the bounds are necessary to obtain the appropriate number of processors in order to achieve the maximum possible speedup.

In the interconnecting topology of the processors, a fully connected switch is assumed. Hence any processor can communicate with all other processors. Given a task graph, a fixed communication pattern can be formed. This means that some message traffic statistics between a pair of processors is observed. Based on this traffic pattern, a topology of interconnection of processors can be designed to achieve an even more efficient execution of an algorithm. This is particularly useful for algorithms which have a fixed communication pattern between processors. With a topology tailored to an algorithm, optimization in the area of efficient message routing can also enhance the execution speed of the algorithm. With the knowledge of the traffic pattern, intelligent routing schemes should be employed for efficient exchange of data enabling the processors to acquire the necessary data faster. This will increase the speedup of the algorithm.

All these thoughts aim at the goal of improving the overall efficiency of dis-

tributed computing. Obviously, there are lots of fruitful areas for further investigation.

## References

1. L. Nagel, "SPICE2 A Computer Program to Simulate Semiconductor Circuits," *ERL Memo No. M520*, May 9 1975.
2. W. T. Weeks et al., "Algorithms for ASTAP - A Network Analysis Program," *IEEE Trans. Circuit Theory*, vol. CT-20, pp. 628-634, Nov. 1973.
3. J. R. Bunch and D. J. Rose, *Sparse Matrix Computations*, Academic Press, 1976.
4. L. O. Chua and P. M. Lin, *Computer-Aided-Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice-Hall, Inc, 1975.
5. A. R. Newton and D. O. Pederson, "Analysis Time, Accuracy and Memory Requirement Tradeoffs in SPICE2," *IEEE Proc. International Symposium on Circuits and Systems*, pp. 8-9, 1978.
6. G. D. Hachtel and A. L. Sangiovanni-Vincentelli, "A Survey of Third-Generation Simulation Techniques," *Proceedings of the IEEE*, vol. 69 No. 10, pp. 1264-1280, Oct. 1981.
7. R. S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, 1962.
8. Ekachai Lelarasmees, *The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits*, PhD Dissertation Department of Electrical Engineering and Computer Sciences University of California Berkeley, Ca 94720, April 1982.
9. O. Wing and J. W. Huang, "A Computational Model of Parallel Solution of Linear Equations," *IEEE Trans. Computers*, vol. C-29, pp. 632-638, July 1980.
10. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, July 1979.



11. Richard M. Fujimoto, *User's Manual for a Multiprocessor Simulator*, Computer Science Division Electrical Engineering and Computer Sciences University of California, Berkeley, CA 94720, Feb 11 1982.
12. Richard M. Fujimoto, "VLSI Communications Components for Multicomputer Networks," *Ph.D Dissertation*, Department of EECS, Computer Science Division, University of California, Berkeley, CA, August 1983.
13. A. Ralston, *A First Course in Numerical Analysis*, McGraw-Hill, 1965.
14. Robert D. Berry, "An Optimal Ordering of Electronic Circuit Equations for a Sparse Matrix Solution," *IEEE Trans. on Circuit Theory*, vol. CT-18, No. 1, Jan. 1971.
15. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
16. Donald E. Knuth, *The Art of Computer Programming Sorting and Searching*, Vol 3, Addison-Wesley, 1973.
17. T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operation Research*, vol. 9 Number 6, pp. 841-848, 1961.
18. T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Comm. ACM*, vol. 17 Number 12, pp. 885-890, Dec 1974.
19. G. K. Manacher, *Production and Stabilization of Real-Time Task Schedules*, 14 No. 3, pp. 439-485, July 1967.
20. W. H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, pp. 1235-1238, Dec. 1975.
21. C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez Jr, "Optimal Scheduling Strategies in a Multiprocessing System," *IEEE Trans. on Computers*,

- vol. C-21 No. 2, pp. 137-146, Feb. 1972.
22. L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc., 1975.
  23. H. S. Stone, "Multiprocessing Scheduling With the Aid of Network Flow Algorithm," *IEEE Trans. on Software Engineering*, pp. 237-245, Jan. 1977.
  24. E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, 1975.
  25. E. Lawler, *Combinatorial Optimization, Network and Matroids*, 1976.
  26. R. Bellman, K. L. Cooke, and J. A. Lockett, *Algorithms, Graphs and Computers*, Academic Press, New York, 1970.
  27. B. A. Farbey, A. H. Land, and J. D. Murchland, "The Cascade Algorithm For Finding All Shortest Distances In A Directed Graph," *Management Science*, vol. 14, pp. 19-28, 1967.
  28. Chung-Hao Jen, "Overhaed and Refigurability Considerations in the Design of Distributed Computing System," *Ph.D dissertation, Department of Electrical Engineering and Computer Sciences. University of California, Berkeley*, 1982.
  29. Kernal Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Trans. on Computers*, pp. 50-56, 1982.

High Speed Recursive Filtering

By

Hui-Hung Lu

B.S. (National Chiao Tung University) 1973

M.S. (University of California) 1980

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:

*David H. Messerschmidt* 11/21/83

Chairman

Date

*Paul R. Gray* 11/29/83

*Lester L. Lau* Dec 1 1983

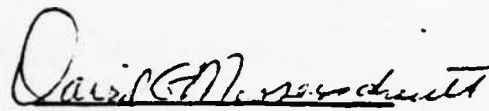
## High Speed Recursive Filtering

Hui-Hung Lu

Ph.D.

Electrical Engineering  
and Computer Sciences

Sponsor:  
National Science Foundation

  
D.G. Messerschmitt  
Chairman of Committee

### ABSTRACT

This dissertation reports on a study of algorithms for the realization of digital infinite impulse response (IIR) filters using multiple programmable elements (PE's). The motivation is to increase the sampling rate which can be achieved with a VLSI implementation of a digital filter. This is best achieved utilizing parallelism since VLSI will provide dramatic increases in hardware complexity with much more modest increases in speed.

The algorithms presented can be classified into two categories, single-input single-output (SISO) and multi-input multi-output (MIMO) filters. Included in the SISO category are the well known cascade and parallel forms, systolic arrays, and Barnwell's algorithm. The MIMO filter is also called a block filter since the input samples are divided into blocks and processed by vector and matrix operations.

The theory as confirmed by simulation results indicate that block filters can achieve a much higher sampling rate than SISO filters, at the expense of a larger amount of hardware. In fact, it is shown that the sampling rate achieved can become arbitrarily large as hardware is added, so that the die area and

computational resources on a chip become the sole limitation on sampling rate, not the speed of the hardware. Block filters achieve their increase in speed by the addition of PE's, as well as by increases in the speed of the PE's, and are therefore well suited to the VLSI technology.

It is further shown that a two dimensional systolic array of PE's can realize the block state structure for an IIR digital filter, making it possible to achieve the aforementioned advantages with only local interconnections of PE's. This meets another important constraint of VLSI, the minimization of expensive global communications.

IIR filters are known to require lower computational rate as compared to a finite impulse response (FIR) filters with approximately the same transfer function. However, block state IIR filters lose their superiority over FIR filters at extremely high sampling rates. An example shows that when the block size exceeds 56 for an elliptic filter, the FIR filter is more advantageous in realizing a similar response.

Block filters are also shown to have excellent properties as far as roundoff noise is concerned. Although the average computation rate is increased, the average output roundoff noise decreases when a single rounding is performed at each internal summing node.

Finally, the various filter structures are compared with respect to their roundoff noise susceptibility, susceptibility to delay in the PE interconnection paths, complexity of PE interconnection, etc.

## ACKNOWLEDGEMENT

I wish to express my sincere gratitude to my research advisor, Professor David G. Messerschmitt, for his valuable inspiration, guidance and encouragement throughout the years of my graduate study. Without his support, none of this would have been possible. Special thanks are due Richard Fujimoto for his assistance in developing the simulation programs. The evaluation of various algorithms presented in this dissertation would be very difficult without these programs. I am also pleased to acknowledge the support from the National Science Foundation.

I am grateful to my parents for their initial financial support and their constant encouragement. Last, but not the least, I wish to thank my wife for her spiritual support and understanding. With her accompany, the graduate study has become more pleasant.

To my parents



## Table of Contents

<b>Chapter 1:</b>	<b>Introduction.....</b>	<b>1</b>
1.1	Impact of VLSI on Signal Processing .....	1
1.2	Simulation Tools .....	3
1.3	Overview .....	7
<b>Chapter 2:</b>	<b>Parallel IIR Filter Design.....</b>	<b>12</b>
2.1	Introduction .....	12
2.2	Cascade and Parallel Forms .....	15
2.3	Systolic Arrays .....	20
2.4	SSIMD Mode .....	23
2.5	Block Processing .....	27
2.6	Block State Space Realization.....	32
2.7	Conclusions .....	42
<b>Chapter 3:</b>	<b>A High Speed Filter Structure.....</b>	<b>43</b>
3.1	FIR Filter Design.....	43
3.2	Real Time Constraint for the IIR Filter Design .....	51
3.3	Direct Implementation.....	54
3.4	Systolic Arrays .....	58
3.5	Some Applications.....	65
3.6	Conclusions .....	73
<b>Chapter 4:</b>	<b>Performance Analysis.....</b>	<b>74</b>
4.1	Speed Limitations.....	74

4.2	Susceptibility to the Transmission Delay .....	86
4.3	Latency Consideration.....	91
4.4	Number of Operations .....	93
4.5	Hardware Usage .....	97
4.6	Conclusions .....	100
<b>Chapter 5:</b>	<b>Effects of Finite Register Length.....</b>	<b>102</b>
5.1	Introduction .....	102
5.2	Stability Consideration.....	106
5.3	Frequency Domain Analysis.....	108
5.4	Time Domain Analysis.....	110
5.5	Optimal SISO Filter Synthesis .....	115
5.6	Conclusions .....	124
<b>Chapter 6:</b>	<b>Simulation Results.....</b>	<b>125</b>
6.1	Introduction .....	125
6.2	Roundoff Noise Simulation.....	135
6.3	Conclusions .....	141
<b>Chapter 7:</b>	<b>Conclusions.....</b>	<b>143</b>
<b>References</b>	<b>.....</b>	<b>146</b>

## CHAPTER 1

### INTRODUCTION

#### 1.1. Impact of VLSI on Signal Processing

As the current semiconductor technology advances, more and more transistors can be put into a single silicon wafer. The trend of this technology development is moving from Large Scale Integration (LSI) toward Very Large Scale Integration (VLSI). This technology improvement will have a great impact on many fields, such as circuit simulation, computer architecture and signal processing etc.. Since the computational power of a chip is directly proportional to the overall number of transistors on a chip, the VLSI circuit can provide a much more powerful computational capability. This higher computational ability will have a great impact on the signal processing systems since it would allow us to implement much more sophisticated algorithms or to process signals at a much higher sampling rate. However, this high speed signal processing cannot be obtained by implementing existing algorithms directly in VLSI, since the speed of VLSI will not increase as rapidly as the complexity. Hence, the first implication of VLSI is that in order to effectively utilize the high density chips, parallel algorithms should be employed.

VLSI is achieved by scaling down the size of the devices, with an attendant reduction in power supply voltage to maintain reliability. This lower supply voltage and smaller devices severely restricts the capability to send data over long wires. Hence, although computation and control functions are relatively plentiful in VLSI, communications is expensive. Communications here means data transmission between different parts of the same chip as well as between chips. Together with the parallel processing requirement, this implies that the most

desirable algorithms for achieving maximum performance in a VLSI system are those which achieve increased parallelism with a minimum communication among the parts of the algorithm. This is why the structures like systolic arrays are attractive in VLSI applications.

Since digital circuitry suffers less from the scaling than analog circuitry and digital processing of signals can usually realize more sophisticated algorithms than analog processing, the trend of signal processing systems is moving toward digital processing. This dissertation deals with the new algorithms in digital signal processing systems specifically for the implementation of digital IIR filters.

With conventional digital signal processing algorithms, the throughput rate depends heavily on the chip speed, since the input signal is usually processed in serial. In order to achieve a higher throughput rate, some modification of the existing algorithms has to be made to exploit this high chip density. The above characterization of VLSI suggests that structures like parallel or pipelining would be very efficient in processing high speed signals using relatively low speed hardware.

The Fast Fourier Transform (FFT) is a good example to show the importance in modifying the current algorithms and the tradeoff between the overall speed and the hardware requirement. The FFT has been considered as a very efficient algorithm to compute the Discrete Fourier Transform (DFT) because it requires much less computation. However, the complex data flow pattern in the FFT[1] prohibits us from utilizing the parallel processing technique, since the mutually dependent data manipulations make it difficult for a VLSI design. Although FFT has a pipelining structure in nature, the butterfly connection among cascading stages requires a lot of wires for interconnection. A better algorithm to maximize the throughput would be to compute the DFT directly as a matrix-vector mul-

multiplication. Many structures, which use extensive pipelining and requires simple interconnections among processing elements, are known to be very efficient in performing this multiplication.

The DFT example suggests that we reconsider the existing fast algorithms in digital signal processing systems. It also implies that in order to increase the throughput rate, we should try to find algorithms which effectively exploit parallelism rather than necessarily reduce the number of multiplications. Although most existing signal processing algorithms are in a serial form, the potential of parallelism makes high speed algorithms possible in VLSI circuits. What has to be done is to find algorithms which can maximize the parallelism.

## 1.2. Simulation Tools

Simulation programs are necessary to examine the performance of the parallel algorithms to be implemented. The simulation is important because it allows us to avoid building the hardware and because simulation can minimize the hardware design time. In this dissertation, the performance as measured by the speedup compared to the uniprocessor case. The data transmission requirements will be determined by simulation. To satisfy this demand, two simulation programs, both written in 'C', have been developed and are running on a VAX-11/780 machine. SIMON ( Simulator of MultiprOcessor Networks[2] ), is a discrete-time and event driven simulation program, which executes a set of tasks as if they are executed simultaneously. Actually, SIMON can simulate any program running on a multiprocessing system for which the interconnection topology is specified. This simulator has also been successfully applied measuring the performance of algorithms for concurrent circuit simulation[3] which is quite different from those in the signal processing systems.

The other simulator, BLOSIM (BLOck SIMulator[4]), is a discrete-time time-driven simulation program. It is used to simulate sampled-data systems only

and hence is not so general as SIMON. However, it is very efficient for simulating systems such as digital filters and other signal processing systems with a regular sampling rate.

The structure of a general multiprocessing system can range from a collection of general programmable processors to to an interconnection of dedicated hardware elements, such as array processors, systolic arrays etc.. A general multiprocessing system is usually modeled as a combination of calculating and switching processors. The switching processors are devoted to the data transfer among computing processors only; hence, they do not contribute to the speedup directly. The advantage of this model is that any network can be simulated by inserting a proper switch network. On the other hand, the dedicated structure can perform a specific task in a more efficient fashion.

#### 1.2.1. SIMON

The simulator consists of three components (See Figure 1-1), the application program, the simulator base, and the switch model. The application program consists of a number of tasks, or equivalently processes, which are filter routines in our case. The simulator kernel time-multiplexes execution of the tasks on the host computer. The kernel also keeps track of time for each task to ensure that interactions among tasks are simulated in the proper time sequence. Each task has its own clock which advances as the task executes. Finally, the switch model provides a fixed virtual circuit communication mechanism among the tasks and simulates message passing between processors.

The most important features of the simulator are summarized as follows:

- (1) SIMON provides the timing statistics of each task, which includes the running time as well as the blocked time, once the processor speed is given. If

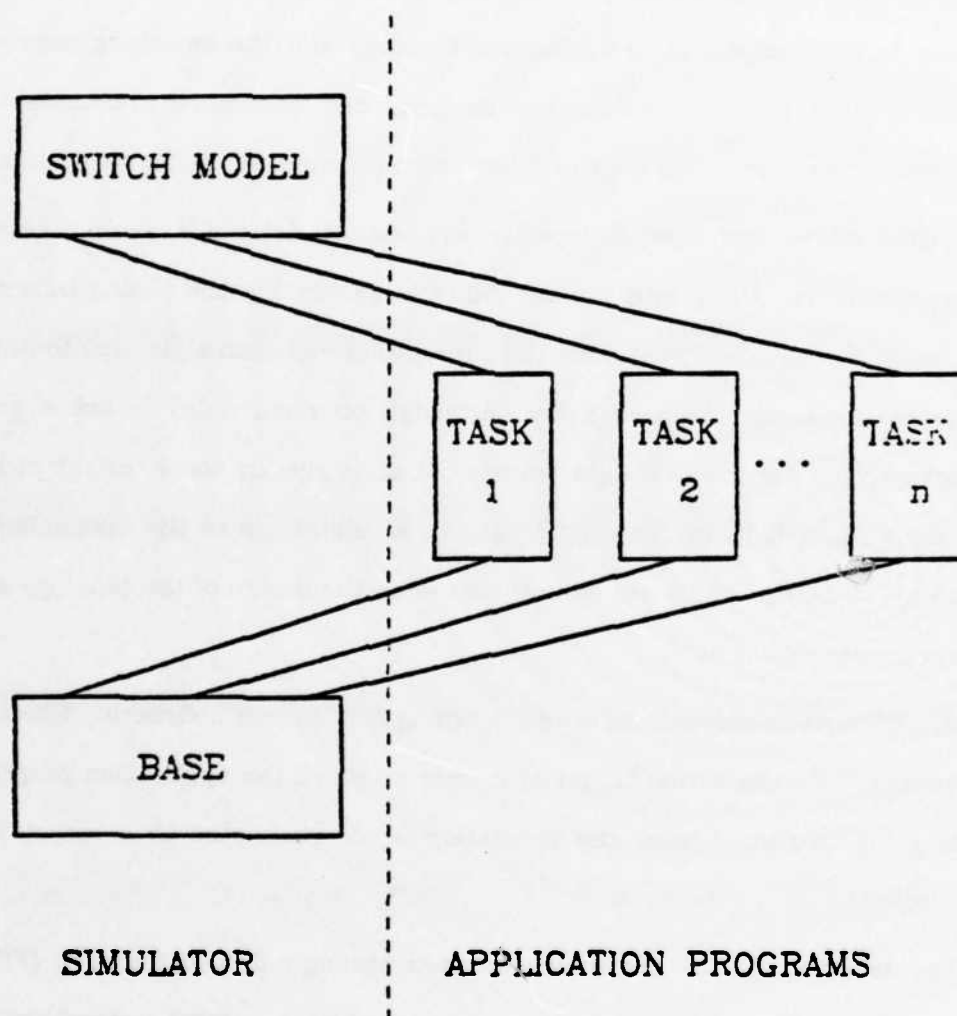


Figure 1-1 Program Structure of SIMON



the processor speed is not specified, a VAX machine is assumed by default. This information is useful in measuring the speedup and in giving insight in the processor usage efficiency.

- (2) SIMON also shows the average traffic statistics for each communication link. Upon request, it can even provide temporal traffic statistics. This information is very helpful in designing the topology and the switching network, since using this information the designer can dynamically allocate the route for each message transmission to avoid the heavy traffic congestion.
- (3) SIMON allows the user to specify a constant delay for each message transmission. Along with the capability of generating the timing information, it can effectively measure the susceptibility of each algorithm to inter-processor communication delay. Although constant delay is not a good assumption for the real data transfer, it does give us some insight of the susceptibility of the throughput rate of an algorithm to the transmission delay. A better model will be available when the design of the topology and switch network is done.
- (4) SIMON permits the user to create a timing file to force instruction times to approximate the actual target processor on which the application program is going to run. Hence, the simulation is not restricted to a target VAX machine only.

The tasks in SIMON are interconnected through first-in first-out (FIFO) buffers. Connection of two tasks is accomplished by a naming convention: an output FIFO will be connected to all the input FIFO's with the same name. Data transfers between a processor and an input or an output FIFO are achieved by calling two function, namely `get()` and `put()`, respectively.

### 1.2.2. BLOSIM

BLOSIM is efficient for simulating systems which operate on data at regular time intervals. Hence, it can effectively simulate sampled data systems, such as digital filters. It can even accommodate system with different and asynchronous sampling rates present simultaneously. The user partitions the system into small pieces implementing elementary parts of the system, each piece called a "block". The user also provides a program which defines the topology of interconnection of those blocks. The actual interconnection is handled by BLOSIM in a similar fashion as SIMON. The difference is that the switch model is not required and the FIFO's are usually of finite length, whereas in SIMON, we can assume infinite length FIFO's. If only the function of an algorithm is simulated and the completion time of each task is not needed, BLOSIM is more efficient than SIMON.

### 1.3. Overview

The signal processing system can be implemented on either off-the-shelf programmable chips or a dedicated hardware circuit. Each part of the circuit can be a very complex microprocessor with memory or a very simple logic circuit such as an ALU plus some registers. In the later chapters, they will be referred to as Processing Elements (PE's).

A signal processing system is realized on an interconnected set of these PE's, which can be structured in any configuration. The structure can vary from a very general one, which is referred to as a multiprocessor structure, to a well-defined structure, such as tree, ring, pipelining and parallel structures etc..

As mentioned before, pipelined and parallel structures are very common in signal processing and very efficient for VLSI applications. The simple pipelined structure is a linear array of interconnected PE's. Each PE fetches the output

from the previous one and sends its output to the next PE when it finishes its computation. Therefore, this structure can be viewed as an array with each PE working on successive input samples. Its structure will become clear in chapter 2 when the cascade form of a digital filter is discussed.

On the other hand, a parallel form also composes a set of PE's. However, all these PE's operate on the same input sample simultaneously. In the typical example of a parallel digital filter design, which will be shown in the next chapter, the output of each PE is sent to a common place to be summed up.

### 1.3.1. Objectives

With the above structures in mind, we will demonstrate various combinations of algorithms and structures which can efficiently utilize the inherent parallelism in digital signal processing systems. Several algorithms for the parallel realization of specifically digital IIR filters will be presented in this dissertation. An IIR (Infinite Impulse Response) digital filter has an infinite length for its impulse response and usually is realized by the recursive technique. Digital filters are widely used in signal processing and control systems. Furthermore, the recursive aspect of IIR features is very representative of signal processing systems and presents a problem for parallel processing. Hence, designing the IIR digital filters in a parallel form is a good application to give insight into the realization of parallelism in signal processing systems.

On the other hand, FIR digital filters are easy to realize in a parallel form. Suppose a filter is realized by the direct form. This filter can be duplicated with as many sections as necessary, with each section calculating one output sample. This can be shown as in Figure 1-2. Due to its feedback free phenomenon, no communication among these sections is required.

Since most signal processing systems require real time, it would be desir-

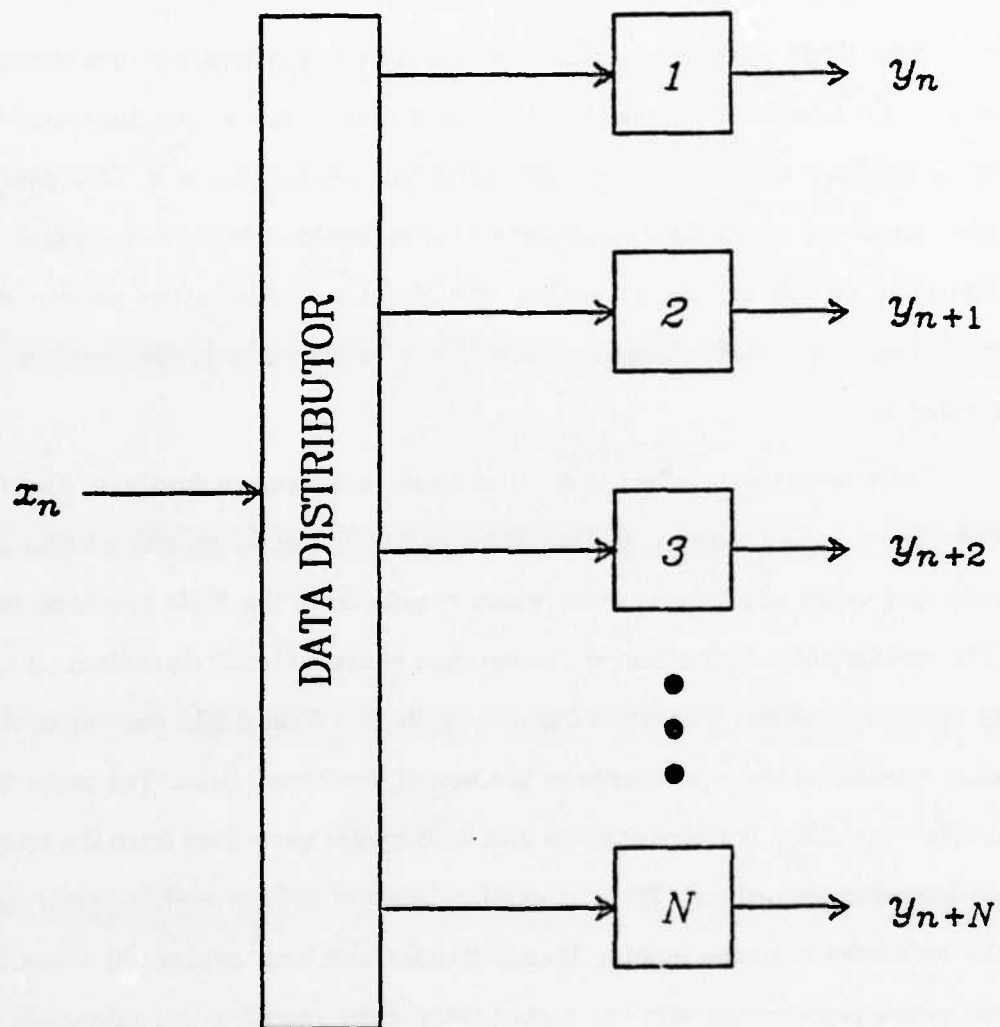


Figure 1-2 Parallel FIR Filter Structure

able to find algorithms such that real time filtering is achievable no matter how fast the input sampling rate or how slow the processor speed is. For those non real time signals, such as geophysical signal processing, high throughput rate is usually desired. Thus, exploiting parallelism to realize a high speed digital filter is our main objective.

Once these parallel algorithms are available, efficient parallel structures should be developed to realize these algorithms. Since the interconnecting wires, whether within a chip or among chips, are expensive in VLSI design, a good structure should have interconnection as localized as possible. Besides the limitation on the driving capability, the global communication pattern would also increase the message transmission delay, which will degrade the speed performance.

Finite word length effect is another important issue to consider. There are two major considerations in this area. One is the sensitivity of the filter response to the coefficients error which results from the finite precision of the filter coefficients. This error will sometimes cause a stability problem. If there is a pole close to but still within the unit circle, the filter might become unstable after quantizing the coefficients in the actual implementation. The other finite precision problem is roundoff noise and limit cycles generated from the internal arithmetical operations. These operations usually require higher precision for the outcomes than the inputs. Roundoff noise and limit cycles will occur if all values are represented with the same precision by rounding the outcomes. The difference between roundoff noise and limit cycles is that the former results from uncorrelated noise sources while the later from the correlated noise sources. Dynamic range is another effect which is closely related to the roundoff noise. Thus, a good parallel algorithm, in addition to speedup, should also improve or at least not adversely impact the effect of finite word length.

Before actually implementing the filter, the user has to fully understand the range of algorithms and then choose the most appropriate one. This increases the design time. Programming a large number of PE's, if they don't have identical programs, is also involved. Therefore automatic program generators for some algorithms are also developed to help design parallel digital filters. Once the filter response, input sampling rate and chip speed are given, the program generator is able to first choose the appropriate structure, to generate coefficients, and then to write the programs running on SIMON.

### 1.3.2. Scope of the Dissertation

In the next chapter, various parallel algorithms and structures for the realization of IIR digital filters are presented. These algorithms can be divided into two categories; namely, single-input single-output (SISO) and multi-input multi-output (MIMO) systems. A highly efficient structure which uses extensive pipelining and multiprocessing is described in chapter 3. This high speed structure is realized in the state space domain rather than in the usual I/O domain. The performance of all the algorithms mentioned in chapters 2 and 3 is analyzed and compared in chapter 4. The performance comparison concerns the speed limitation, transmission delay effect on speed, latency consideration, computational rate between FIR and IIR filters and the efficiency of the processor usage. An important filter design parameter, roundoff noise, is discussed for the block state filters in chapter 5. In chapter 6, simulation results are shown which verify the analysis described in chapters 4 and 5.

## CHAPTER 2

## PARALLEL IIR FILTER DESIGN

Unlike Finite Impulse Response (FIR) filters, it is not obvious how we can design a high speed IIR filter utilizing parallelism. The difficulty arises because the output from an IIR filter depends not only on the current and previous input samples but also on the previous output samples. This feedback would seem to put an upper bound on the speed of operation. We will concentrate here on the IIR filter implementation only; however, all the algorithms presented below can also be applied to FIR filters as a degenerate special case.

## 2.1. Introduction

In this and the following chapters, assume the filter to be implemented is characterized by difference equation (2-1), unless otherwise specified,

$$y_n = \sum_{i=1}^N b_i y_{n-i} + \sum_{i=0}^M a_i x_{n-i} \quad (2-1)$$

where the coefficients are real numbers. Taking the  $z$  transform on (2-1), the transfer function is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^M a_i z^{-i}}{1 - \sum_{i=1}^N b_i z^{-i}} \quad (2-2)$$

There are two interesting cases in equation (2-1). One, the finite impulse response (FIR) filter corresponds to  $N=0$ . The other, the infinite impulse response (IIR) filter, corresponds to  $N \geq 1$ ,  $N > 0$  and  $b_1 \neq 0$ . The tradeoff between these two cases is roughly that to approximate a given desired response, the FIR filter requires a much larger order  $M$  and hence a larger computation rate (as measured by multiplies and adds per output sample). However, when the sam-



pling rate at which a filter can be implemented for a given speed of hardware is considered, the FIR filter appears at first examination to be faster because of the natural way in which parallelism can be exploited.

Besides duplicating a filter section implementing equation (2-1) as shown in Figure 1-2, another method for achieving speed with an FIR filter is simply to calculate a vector of  $L$  successive output samples in parallel. To see this, define output and input vectors of  $L$  successive samples as

$$Y_n = [y_{nL}, y_{nL+1}, \dots, y_{(n+1)L-1}]^T \quad (2-3a)$$

$$X_n = [x_{nL}, x_{nL+1}, \dots, x_{(n+1)L-1}]^T \quad (2-3b)$$

where  $n$  denotes the block number. If we take  $L \geq M$  for an FIR filter, then it is easy to see that the filter can be represented as

$$Y_n = AX_n + BX_{n-1} \quad (2-4)$$

where  $A$  and  $B$  are appropriate  $L \times L$  matrices. The form of the filter given by (2-1) we refer to as SISO (single-input single-output) and (2-4) is referred to as MIMO (multiple-input multiple-output). The MIMO system is shown schematically in Figure 2-1. A detailed discussion of the realization of both SISO and MIMO FIR filters will be given in the next chapter.

Similar to FIR filters, the algorithms of IIR filters can also be divided into SISO and MIMO. In this chapter, various parallel algorithms for realizing equation (2-1) with  $N \geq M$  will be presented. The best known cascade and parallel forms, which will be used as references and compared to the other structures, will be briefly explained in the first section. The remaining parallel algorithms will be presented subsequently. Their performance as measured by the overall execution time with and without the message transmission delay will be analyzed in the following chapters and simulation results will also be presented to verify our analyses.

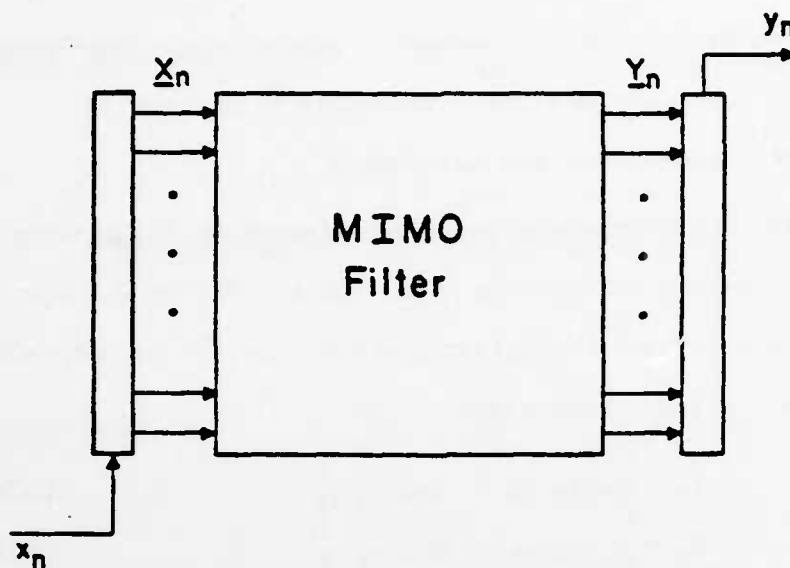


Figure 2-1 Block Diagram of MIMO Filters

A filter with  $M > N$ , can be realized with an FIR filter cascaded with an IIR filter which satisfies the above criteria. Equation (2-2) then becomes

$$H(z) = \sum_{i=0}^{M-N} c_i z^{-i} \frac{\sum_{i=0}^N d_i z^{-i}}{1 - \sum_{i=1}^N b_i z^{-i}}$$

where

$$a_i = \sum_{j=\max(0, i-M-N)}^{\min(i, N)} d_j c_{i-j}$$

Since the parallel implementation of the first term in the right hand side is straightforward, the greatest effort will be devoted to the implementation of the second term, which is an IIR filter.

Sections 2-2, 2-3 and 2-4 will be devoted to the algorithms implementing SISO filters. In Section 2-2, the well-known Cascade and Parallel forms will be treated. A special architecture, systolic array, will be discussed afterwards. Finally, a high speed Single Instruction Multiple Data mode algorithm will be dealt with in Section 2-4. MIMO filters will be discussed in Sections 2-5 and 2-6.

Before discussing the cascade and parallel forms of a digital IIR filter, a brief definition of *realization* and *structure* of a filter will be given, because they are going to be used over and over in the following discussion. *Realization* of a digital filter is any configuration of a hardware or a set of programs implementing a set of arithmetic operations and delay elements which can achieve the transfer function as in (2-2). *Structure* is a special circuit configuration or a specific sequence of instructions in a program to realize a filter. For example, direct, cascade and parallel forms are three different *structures* of a digital filter. However, all these *structures* can be used to *realize* the same filter equation.

## 2.2. Cascade and Parallel Forms

In conventional filter design, the most popular structures are the cascade and parallel connection of 2<sup>nd</sup> and/or 1<sup>st</sup> order filters. They are popular not only because they are modular, but also because they are insensitive to roundoff noise. Their modularity makes them easily expandable to any size and their insensitivity guarantees a large dynamic range with a minimum number of bits or accuracy in the architecture.

### 2.2.1. Cascade Form

For a filter with real coefficients, we can factor the numerator and denominator of equation (2-2) into a product of 1<sup>st</sup> and/or 2<sup>nd</sup> order polynomials while keeping all the coefficients real. Thus, equation (2-2) can be rewritten as

$$H(z) = \prod_{i=1}^P \frac{N_i(z)}{D_i(z)} \quad (2-5)$$

Each term in the product is the transfer function of a second order filter for a complex conjugate pair of poles or a first order filter for a real pole. Whenever possible, we can also combine two real poles to form a second order filter. Figure 2-2 shows the structure of this form.

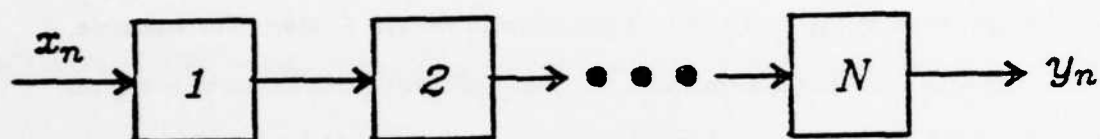


Figure 2-2 Cascade Form Realization of an IIR Digital Filter

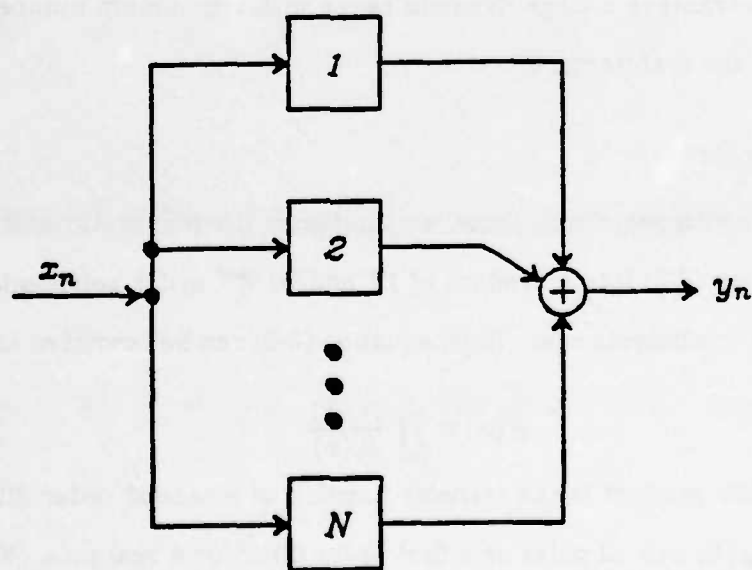


Figure 2-3 Parallel Form Realization of an IIR Digital Filter

The problems left are the pairing of poles and zeros and the ordering of those second and first order filters. Although neither pairing nor ordering will affect the overall transfer function and the overall speed, they will usually affect the roundoff noise behavior[5].

### 2.2.2. Parallel Form

To realize a filter in a parallel form, we have to first obtain a partial fractional expansion form of the original transfer function and then realize each term independently. The output from each section is sent to an adder to sum up in order to get the final result. If all the poles are simple, equation (2-2) can be expressed as

$$H(z) = d + \sum_{i=1}^P \frac{N_i(z)}{D_i(z)} \quad (2-6)$$

The numerator of each term in the summation is a polynomial with degree one less than that of the denominator. For a complex conjugate pair, the denominator is a polynomial of degree 2, while for a real pole it is of degree 1.  $d$  is non-zero if  $M=N$  and zero if  $M<N$ . This constant term can also be distributed to each filter section. Then, equation (2-6) becomes

$$H(z) = \sum_{i=1}^P \left[ d_i + \frac{N_i(z)}{D_i(z)} \right] \quad (2-7)$$

where  $\sum_{i=1}^P d_i = d$ . The structure of this form is shown in Figure 2-3.

The conclusion that each term in the summation in (2-6) is either a 1<sup>st</sup> or a 2<sup>nd</sup> filter is based on the assumption that all the poles are simple. If any poles with multiplicity greater than one exist, some denominators must have a degree higher than 2 for complex pole pairs. Suppose an 8<sup>th</sup> order filter, which has two simple complex conjugate pairs of poles and a multiple order complex conjugate pole with multiplicity 2, is realized in a parallel form. The transfer function of this filter can be represented by a summation of three terms as shown in (2-8).

$$H(z) = \frac{N_1(z)}{(z-p_1)(z-p_1^*)} + \frac{N_2(z)}{(z-p_2)(z-p_2^*)} + \frac{N_3(z)}{(z-p_3)^2(z-p_3^*)^2} \quad (2-8)$$

where the first and the second terms are second order filters and the third term is a fourth order filter. The filter structure can be drawn as in Figure 2-4a. where block 3 is a fourth order filter. It is obvious that the overall speed is governed by this fourth order filter instead of a second order section. If transforming the third term in (2-8) into a product of two second order filters as follows

$$\frac{N_3(z)}{(z-p_3)^2(z-p_3^*)^2} = \frac{N_{31}(z)}{(z-p_3)(z-p_3^*)} \times \frac{N_{32}(z)}{(z-p_3)(z-p_3^*)}$$

where  $N_{31}(z)N_{32}(z) = N_3(z)$ , the overall speed again depends on a 2<sup>nd</sup> order filter, if delay lines are introduced after blocks 1 and 2. The overall structure is shown in Figure 2-4b. The dotted squares are delay elements and the solid squares are 2<sup>nd</sup> order sections.

An alternative method is to transform (2-8) into a product of a 6<sup>th</sup> order filter and a 2<sup>nd</sup> order filter, which can be represented by equation (2-9).

$$\begin{aligned} H(z) &= \frac{M_1(z)}{(z-p_1)(z-p_1^*)(z-p_2)(z-p_2^*)(z-p_3)(z-p_3^*)} \times \frac{M_2(z)}{(z-p_3)(z-p_3^*)} \\ &= \left[ \frac{M_{11}(z)}{(z-p_1)(z-p_1^*)} + \frac{M_{12}(z)}{(z-p_2)(z-p_2^*)} + \frac{M_{13}(z)}{(z-p_3)(z-p_3^*)} \right] \\ &\quad \times \frac{M_2(z)}{(z-p_3)(z-p_3^*)} \end{aligned} \quad (2-9)$$

A structure that realizes (2-9) is shown in Figure 2-5 with each square representing a 2<sup>nd</sup> order filter. This structure is very important in achieving high speed block filters which are insensitive to the message transmission delay and will be shown in Chapter 3.

Applying the above technique to all the multiple poles, (2-7) can be rewritten as

$$H(z) = \prod_{i=1}^K \sum_{j=1}^{P_i} \left[ \frac{N_{ij}(z)}{D_{ij}(z)} + d_{ij} \right] \quad (2-10)$$

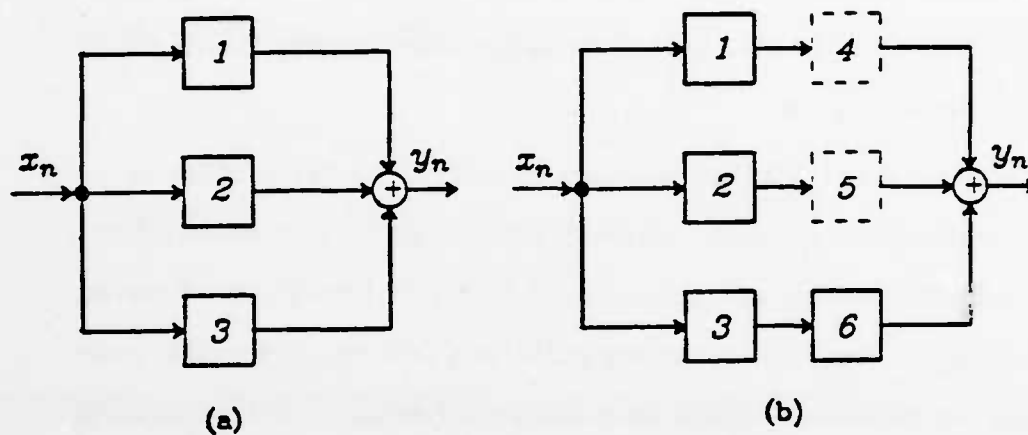


Figure 2-4: Parallel Form Realization with Multiple Order Poles (a) Different Order with Each Subfilter (b) With Inserted Delays on Low Order Subfilters

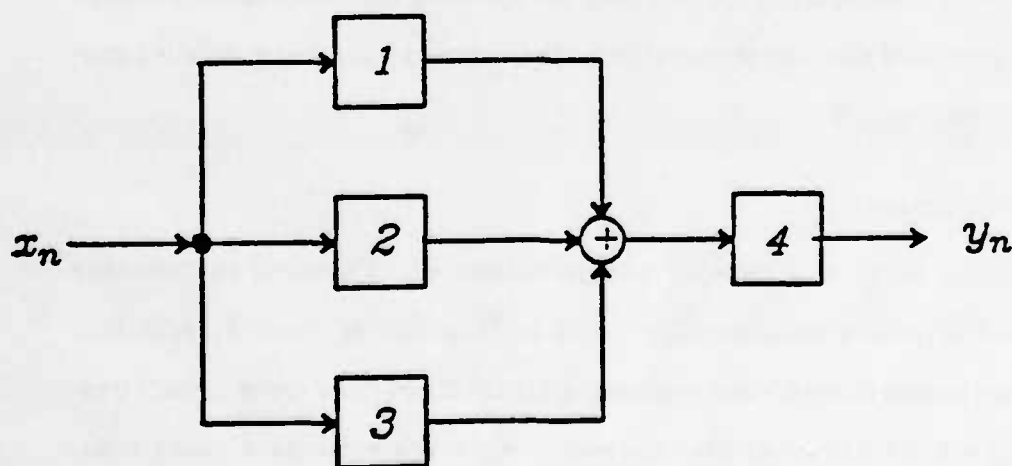


Figure 2-5 Cascade-Parallel Form Realization



where  $\frac{N_j(z)}{D_j(z)}$  is the transfer function of a  $2^{nd}/1^{st}$  order filter. And  $K$  is the maximum multiplicity among all the multiple poles and  $P_j$  is the number of subfilters in the  $j^{th}$  stage. Thus, the minimum number of stages required to achieve this second order filter connection is the maximum multiplicity among all the multiple order poles.

In Figures 2-2 and 2-3, the structure of each  $2^{nd}$  order subfilter is not shown, because there are many different structures. Some canonical forms require less computation and hence result in faster throughput. However, roundoff noise and coefficient sensitivity might be a problem. Some other realizations can assure us low roundoff noise, but usually require more computation and hence lower speed. On the other hand, since lower roundoff noise implies fewer bits are required to represent each number, each computation takes less time. This might compensate the speed degradation problem. Therefore, a designer has to consider the tradeoff between roundoff noise and speed. However, when considering the actual implementation, lower roundoff noise implies fewer bits are required to represent each number. Hence, lower roundoff noise also implies less computation which may compensate the lower speed resulted from the increased number of operations. These issues will receive detailed consideration in Chapter 5.

### 2.3. Systolic Arrays

A systolic array is a network of interconnected PE's which rhythmically computes and passes data through its PE's. It is one alternative realization which must be considered when realizing a digital filter. This array usually connects only a few types of simple processors which are sometimes called cells. The data flow pattern through these cells is usually simple and regular so that cells can be connected by a network with local and regular interconnections.

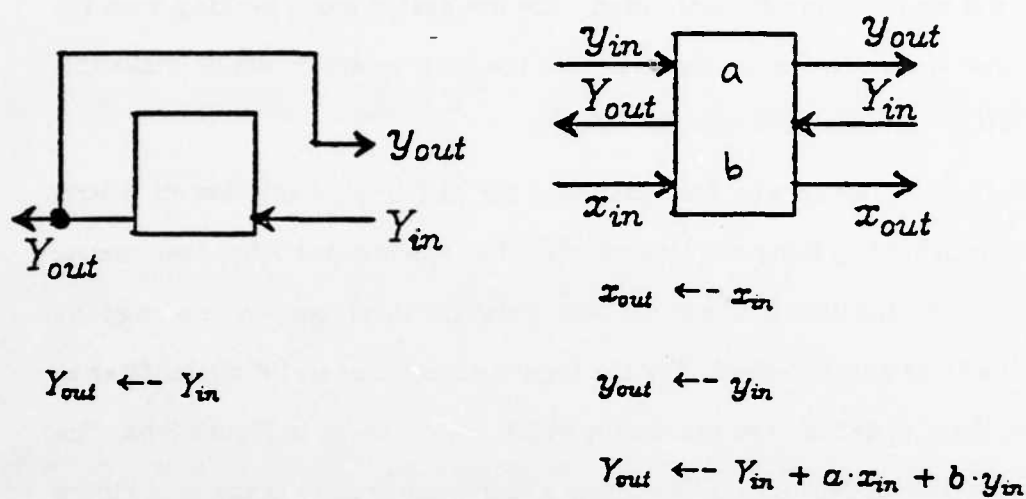
This array can sustain very high throughput rate because it uses extensive pipelining and multiprocessing. Furthermore, the architecture of every cell is very simple and each cell is efficiently used. The low design cost resulting from the replication of identical cells, together with the local interconnection make this architecture extremely suitable for VLSI design.

The systolic array was first proposed for the implementation of matrix operations in VLSI by Kung and Lieserson[6]. For applications other than matrix operations, the function of the basic cell(s) and the interconnection among the cells have to be characterized. For the implementation of an IIR digital filter as in (2-1), Kung[7] defined two basic cells which are drawn as in Figure 2-6a. The interconnection of cells for implementing a fourth order filter is shown in Figure 2-6b. The implemented filter can be represented by the following equation

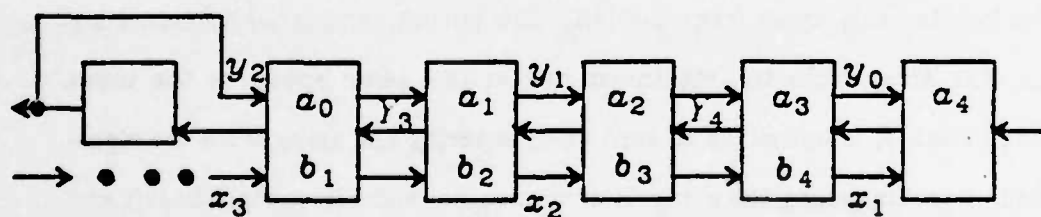
$$y_n = \sum_{i=1}^4 b_i y_{n-i} + \sum_{i=0}^4 a_i x_{n-i}$$

Each cell in the array of Figure 2-6b contains two coefficients. One of them multiplies the  $x$  input and the other one multiplies the  $y$  input. The input samples,  $\{x_n\}$ , enter this array from the left. The output samples,  $\{Y_n\}$ , traveling through this array from the rightmost cell at the same speed as the input sequence. Each  $Y_i$  is initialized to zero when entering the array from the rightmost cell. Marching along the array, it accumulates terms from right to left and eventually obtains its final value when reaching the left-most cell. The final value of  $Y_i$  is also fed back to the array for use in computing  $Y_{i+1}$  to  $Y_{i+4}$ . The feedback sequence is labeled as  $\{y_n\}$  to distinguish from the output samples.

Each cell except the first one communicates only with its right and left neighbors, while the first cell talks only to its right neighbor. No global communication is required anywhere in this architecture. Since every two adjacent samples of both  $x_i$  and  $y_i$  sequences are separated by two clock cycles to ensure that each  $x_i$  meet every  $y_i$ , it is clear that only half of the cells in the array are



(a)



(b)

Figure 2-8: Linear Systolic Array for Realizing IIR Filters (a) Function of Basic Cells (b) Structure of a Fourth Order IIR Filter

active at any given time. Thus, we can combine two adjacent cells into one so as to fully utilize the hardware resources.

As for FIR filters, their implementation is equivalent to the realization of finite convolutions. Kung[8] mentioned several different ways of realizing finite convolutions with systolic approaches. One type of realization uses the same structure as in Figure 2-6b, except that each cell has only one coefficient and no feedback links exist. Further, the input samples travel twice as fast as the output samples. The realization of FIR filters will be discussed in detail in the next chapter.

#### 2.4. SSIMD Mode

Yet another possibility for digital filter realization in the Skewed Single Instruction Multiple Data (SSIMD) mode, in which exactly the same arithmetic operation is executed on a set of identical processing elements. The starting time of each PE is skewed by a fixed amount to work on successive input samples. This implementation can be applied to any signal flow graph including multiplication, addition and delay. Barnwell[9] decomposed equation (2-1) into a set of two equations as

$$r_n = \sum_{i=1}^N b_i r_{n-i} + x_n \quad (2-11a)$$

$$y_n = \sum_{i=0}^M c_i r_{n-i} \quad (2-11b)$$

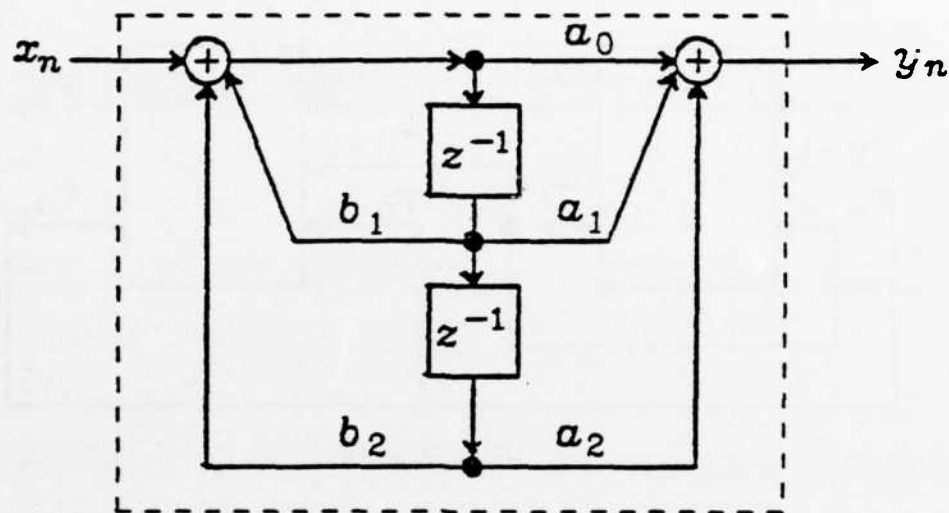
This is equivalent to decomposing the filter into an all pole filter followed by an all zero filter. Both equations are executed on every PE to generate not only the output sample  $y_n$  but also the output sample  $r_n$  from the all pole section. PE's are skewed in time to work on inputs and outputs of different time indices. The delayed versions of  $r_n$  in each PE are not computed internal to that PE, but are supplied from other PE's executing the same code. Therefore, each PE has to fetch not only the input samples but also the intermediate outputs from some

other PE's.

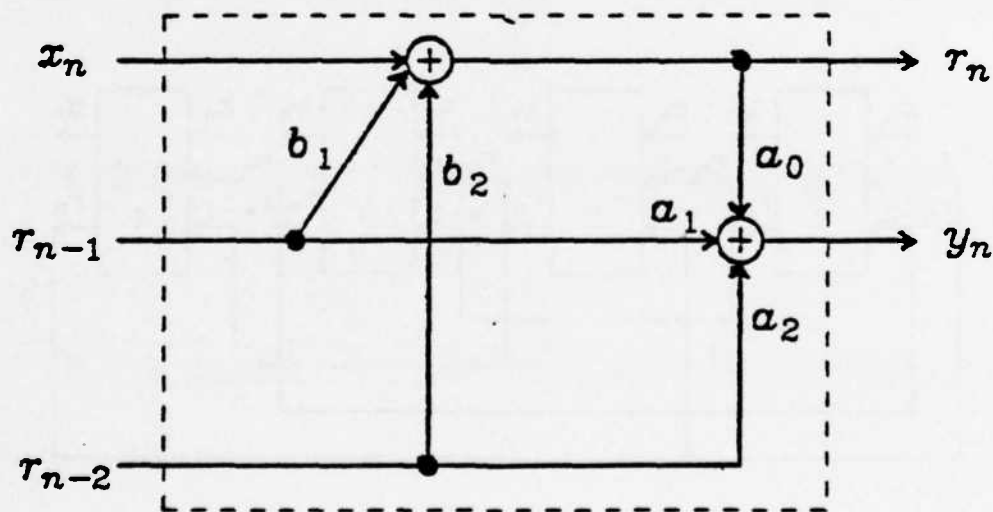
The fundamental concept of this implementation is illustrated by a second order filter example shown in Figures 2-7. In this example, the second order direct form filter of Figure 2-7a is implemented on a single PE according to equations (2-11) where  $r_{n-1}$  and  $r_{n-2}$  are computed internally in the previous cycles. Figure 2-7b shows the function of one PE in a multiprocessor realization, where  $r_{n-1}$  and  $r_{n-2}$  are extra input samples and  $r_n$  is an extra output sample. Figures 2-8 shows an example of implementing this second order filter on two and four PE's.

In Figure 2-8a, a two processor realization is illustrated. The main reason that multiprocessing is possible is that, for PE 1, even though the value of  $r_{n-1}$  must be available before we can obtain  $r_n$ , it is not necessary for it to be available before the computation of  $r_n$  is started. What is required, rather, is that the value of  $r_{n-1}$  must be available before it is used by PE 1. Hence, PE 1 may start computing before  $y_{n-1}$  and even  $r_{n-1}$  are available. The availability of  $r_{n-2}$  is not a problem, since it is computed by the same PE in the previous cycle; hence, it is always available when we start computing  $r_n$ . On the other hand, for a four processor implementation as in Figure 2-8b, PE 1 may start computing as soon as it is guaranteed that both  $r_{n-2}$  and  $r_{n-1}$  are available when they are needed. If the availability of these two values gives different constraints on the starting time, we have to choose the later starting time for an obvious reason. This argument should be easily extended to filters of any order. For an  $N^{\text{th}}$  order filter, the availability of  $r_{n-1}$  through  $r_{n-N+1}$  in each PE has to be considered, and then the latest starting time is chosen.

This implementation has two good features. First, since it is a single instruction multiple data mode, only one program has to be generated for all the PE's. This is a good property especially when a large number of PE's is

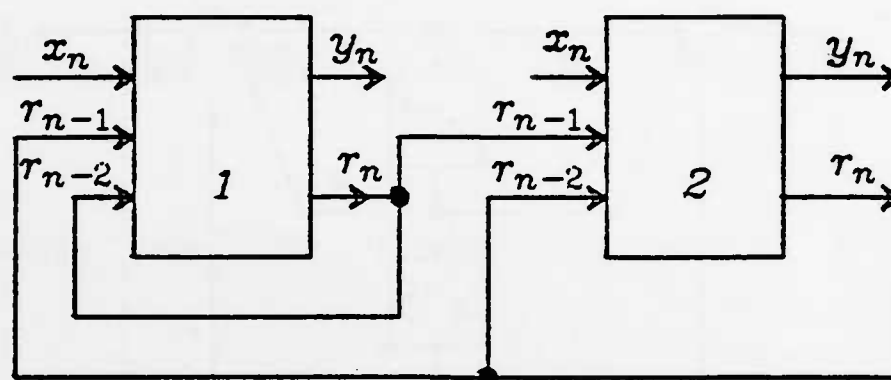


(a)

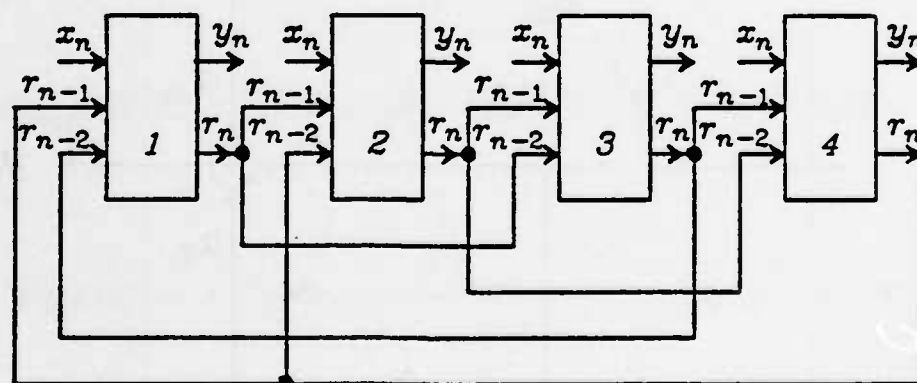


(b)

Figure 2-7: SSMD Structure for IIR Filter Realization (a) Internal Structure of a Single PE (b) I/O Specification of Each PE for Multiprocessing Realization



(a)



(b)

Figure 2-8: Multiprocessing Realization of a 2<sup>nd</sup> Order IIR Filter (a) Two Processor Implementation (b) Four Processor Implementation



involved. The other advantage of this mode is that, the data precedence relations among PE's are automatically maintained by the intrinsic synchrony of the system. However, the irregular and non-local data flow pattern can cause a serious problem in speed performance, as will be discussed in Chapter 4.

All the algorithms mentioned above concerned SISO filter design only, and no MIMO filters have been discussed yet. The processing of the input samples in SISO filters is on a sample-by-sample basis. In the rest of this chapter, effort will be devoted to MIMO filter design. The input samples will be processed by blocks, which is the name block processing named for.

## 2.5. Block Processing

Similar to the FIR filter design, IIR filters can also be realized in a block form as shown in Figure 2-1. This is done by accumulating the input samples in a buffer of size  $L$  and then process these samples simultaneously. Historically, the preliminary motivation behind the use of block processing was the possibility of employing FFT techniques for intermediate computation. Although communication is a serious problem for FFT in VLSI, structures of block implementation exist which are very efficient for multiprocessing. We will discuss an input-output formulation in this section and a state equation formulation will be given in the next section.

Stockham[10] has shown that filters with zeros alone can be synthesized by using the FFT algorithm. Hence, the FIR filters can be realized efficiently by this fast algorithm. Although we once assumed that only recursive techniques can be employed for IIR filters, Gold and Jordan[11] proved that this was not true. They have shown that IIR filters can be synthesized by a combination of three finite convolutions, assuming some initial conditions. Voelcker[12] considered the same problem using the  $z$  transform and showed that recursive filters could be realized by combination of finite convolutions and block feedback. However,

his algorithm introduced additional poles, which might cause a stability problem.

Later, Burrus[13] developed a block feedback structure in time domain based on the matrix representation of convolutions. This algorithm does not cause any stability problems since the extra poles are always located at the origin. Mitra[14] later showed that Gold and Jordan's formulation is actually a particular case of this algorithm. Since Burrus' algorithm is very efficient for parallel processing, a detailed discussion will be given here.

### 2.5.1. Basic Block Feedback

To ease notation, let us consider a third order filter with transfer function as in equation (2-12).

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}{1 - b_1 z^{-1} - b_2 z^{-2} - b_3 z^{-3}} \quad (2-12)$$

In time domain, the filter can be represented in a matrix form as in equation (2-13). Notice that both matrices grow indefinitely until the input samples are exhausted.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -b_1 & 1 & 0 & 0 \\ -b_2 & -b_1 & 1 & 0 \\ 0 & -b_2 & 0 & 1 \\ 0 & 0 & -b_1 & 1 \\ 0 & 0 & -b_2 & -b_1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & 0 \\ a_1 & a_0 & 0 & 0 \\ a_2 & a_1 & 0 & 0 \\ 0 & a_2 & 0 & 0 \\ 0 & 0 & a_0 & a_0 \\ 0 & 0 & a_1 & 0 \\ 0 & 0 & a_2 & a_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} \quad (2-13)$$

If define

$$\begin{aligned} B_0 &= \begin{bmatrix} 1 & 0 & 0 \\ -b_1 & 1 & 0 \\ -b_2 & -b_1 & 1 \end{bmatrix} & B_1 &= \begin{bmatrix} -b_3 & -b_2 & -b_1 \\ 0 & -b_3 & -b_2 \\ 0 & 0 & -b_3 \end{bmatrix} \\ A_0 &= \begin{bmatrix} a_0 & 0 & 0 \\ a_1 & a_0 & 0 \\ a_2 & a_1 & a_0 \end{bmatrix} & A_1 &= \begin{bmatrix} a_3 & a_2 & a_1 \\ 0 & a_3 & a_2 \\ 0 & 0 & a_3 \end{bmatrix} \end{aligned} \quad (2-14)$$

and  $X_n = [x_{3n}, x_{3n+1}, x_{3n+2}]$   
 $Y_n = [y_{3n}, y_{3n+1}, y_{3n+2}] \quad n=0,1,2, \dots$   
 equation (2-13) can be rewritten as in (2-15).

$$\begin{bmatrix} B_0 & 0 & \dots & 0 & 0 \\ B_1 & B_0 & \dots & \dots & \dots \\ 0 & B_1 & \dots & \dots & \dots \\ \dots & \dots & B_0 & 0 & \dots \\ \dots & \dots & B_1 & B_0 & \dots \\ \dots & \dots & \dots & B_1 & \dots \end{bmatrix} \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} A_0 & 0 & \dots & 0 & 0 \\ A_1 & A_0 & \dots & \dots & \dots \\ 0 & A_1 & \dots & \dots & \dots \\ \dots & \dots & 0 & \dots & \dots \\ \dots & \dots & A_0 & 0 & \dots \\ \dots & \dots & A_1 & A_0 & \dots \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ \vdots \end{bmatrix} \quad (2-15)$$

Equating both sides on the row containing  $Y_{n+1}$ , the following recursive equation can be obtained.

$$B_0 Y_{n+1} + B_1 Y_n = A_0 X_{n+1} + A_1 X_n$$

Since  $B_0$  is nonsingular, multiply  $B_0^{-1}$  on both sides and obtain

$$Y_{n+1} = -B_0^{-1} B_1 Y_n + B_0^{-1} A_0 X_{n+1} + B_0^{-1} A_1 X_n \quad (2-16)$$

The conversion from equation (2-13) to (2-15) is true if the size of the four matrices in (2-14) exceeds the filter order. Thus, equation (2-16) holds for any block sizes not smaller than 3.

Equation (2-16) can be applied to filters of any order if we define the input and output vectors as in (2-3) and the four coefficient matrices in (2-17). In (2-17), we also assume that the block size is not smaller than the filter order. The block diagram of this structure is shown in Figure 2-9. Equation (2-16) can represent various types of filters. For an FIR filter,  $B_0$  is an identity matrix and  $B_1$  is a null matrix, hence, the equation becomes

$$Y_{n+1} = A_0 X_{n+1} + A_1 X_n$$

which is equivalent to (2-4). Obviously, this equation is true only if the block size is not smaller the number of taps of the filter.

$$\begin{aligned}
 A_0 &= \begin{bmatrix} a_0 & 0 & \dots & 0 & 0 \\ a_1 & a_0 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{M-1} & a_{M-2} & \dots & \dots & 0 \\ 0 & a_{M-1} & \dots & \dots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots \\ 0 & 0 & \dots & a_1 & a_0 \end{bmatrix} & B_0 &= \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ -b_1 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -b_{N-1} & -b_{N-2} & \dots & \dots & 0 \\ 0 & -b_{N-1} & \dots & \dots & 0 \\ \vdots & \vdots & \ddots & 1 & \vdots \\ 0 & 0 & \dots & -b_1 & 1 \end{bmatrix} \\
 A_1 &= \begin{bmatrix} 0 & a_{M-1} & a_{M-2} & \dots & a_1 \\ 0 & \dots & 0 & a_{M-1} & a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & a_{M-1} \\ \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & \dots & 0 & \dots & 0 \end{bmatrix} & B_1 &= \begin{bmatrix} 0 & -b_{N-1} & -b_{N-2} & \dots & -b_1 \\ 0 & \dots & 0 & -b_{N-1} & -b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & -b_{N-1} \\ \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & \dots & 0 & \dots & 0 \end{bmatrix} \quad (2-17)
 \end{aligned}$$

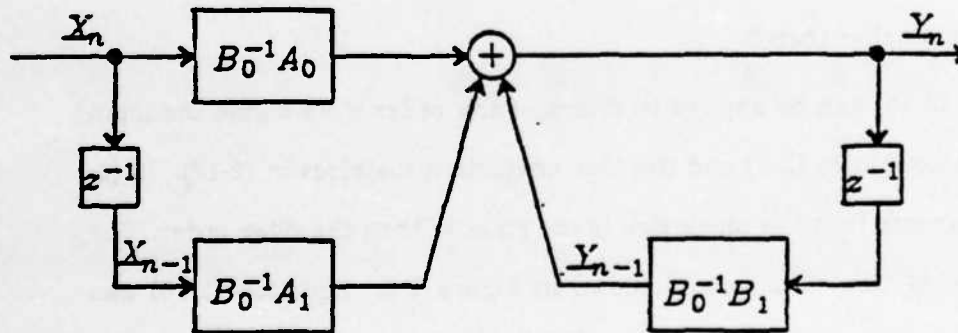


Figure 2-9 Block Diagram of Block I/O Filters

### 2.5.2. Other Structures

In addition to showing two basic structures for implementing equation (2-

16). Mitra and Gnanasekaran derived[15] three other state-structures from the same equation. Generally speaking, all these structures are similar as far as throughput rate is concerned. Hence we will discuss only the structure in figure 2-9.

### 2.5.3. Short Block Lengths

The basic relations become a little more complicated when the block size is less than the order of the filter. For example, if the block size is 2 for the example in the previous section, two more matrices are required. Equation (2-16) can be written as

$$Y_{n+1} = -B_0^{-1}B_1Y_n - B_0^{-1}B_2Y_{n-1} + B_0^{-1}A_0X_{n+1} + B_0^{-1}A_1X_n + B_0^{-1}A_2X_{n-1} \quad (2-18)$$

where the matrices are defined as follows

$$B_0 = \begin{bmatrix} 1 & 0 \\ -b_1 & 1 \end{bmatrix} \quad B_1 = \begin{bmatrix} -b_2 & -b_1 \\ -b_3 & -b_2 \end{bmatrix} \quad B_2 = \begin{bmatrix} 0 & -b_3 \\ 0 & 0 \end{bmatrix}$$

and  $A_0 = \begin{bmatrix} a_c & 0 \\ a_1 & a_0 \end{bmatrix} \quad A_1 = \begin{bmatrix} a_2 & a_1 \\ a_3 & a_2 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & a_3 \\ 0 & 0 \end{bmatrix}$

This can be extended to any block length, if more feedback terms are allowed. In fact, if the block length is reduced to one, equation (2-18) becomes the original scalar difference equation. Hence, the traditional scalar difference equation is a degenerate case of the block difference equation. Furthermore, if factoring and partial fractional expansions are used on the original scalar equation, we can also obtain a parallel or cascade connection of second-order block filters. The block length in each section can also be different from that of the other sections so as to achieve multi-rate digital filters.

### 2.5.4. Pole Locations

Taking the z-transform of equation (2-18), we get

$$zY(z) = -B_0^{-1}B_1Y(z) + zB_0^{-1}A_0X(z) + B_0^{-1}A_1X(z) \quad (2-19)$$

where  $Y(z)$  and  $X(z)$  are the  $z$  transforms of the vectors  $Y_n$  and  $X_n$  respectively. The unit delay  $z^{-1}$  now represents the delay time of a block of samples, which is  $L$  times the delay in the scalar case. If define the transfer function of the multi-input multi-output system as

$$Y(z) = H(z)X(z)$$

it can be obtained directly from (2-19).

$$H(z) = [H_{ij}(z)] = (zI + B_0^{-1}B_1)^{-1}(zB_0^{-1}A_0 + B_0^{-1}A_1) \quad (2-20)$$

where  $H_{ij}(z)$  is the transfer function between the  $i^{\text{th}}$  output sample and the  $j^{\text{th}}$  input sample in a block. From the above equation, it is obvious that the poles of the block filter are the eigenvalues of the product matrix  $B_0^{-1}B_1$ . Mitra and Gnanasekaran[15] verified for the special case  $L = N$  that the eigenvalues of this matrix are the original poles raised to the power  $L$ . This conclusion can also be easily verified by the state space formulation in the following section. However, from equation (2-17), it is also clear that the size of the product matrix grows with the block length  $L$ . We should have  $L$  eigenvalues from this matrix instead of  $N$ , since the size of the feedback matrix  $B_0^{-1}B_1$  is  $L \times L$ . The extra  $L-N+1$  poles in this implementation are actually all located at the origin. This is easily seen by taking a closer look at the matrix  $B_1$ . The left most  $L-N+1$  columns of  $B_1$  are all zero vectors. Thus, the left most  $L-N+1$  columns of the product matrix  $B_0^{-1}B_1$  are also zero vectors, which in turn result in  $L-N+1$  zero eigenvalues.

## 2.6. Block State Space Realization

It is well known[16] that any linear shift invariant system can be represented by state equations. Although requiring more computation per output sample compared to the canonical forms as well as to the forms mentioned above, the state space design can result in very high speed. Furthermore, this high speed can be achieved with very simple interconnection among all the processing elements. A detailed discussion about the speed performance will be

given in the next chapter. In this section, a detailed derivation of the block state equation from the scalar transfer function will be given.

These equations can tell us not only the input-output relationship but also the internal data flow. This is especially beneficial for roundoff noise analysis, since roundoff noise depends heavily on the structure of the filter. In a later chapter, we will derive the roundoff noise power at a filter output in terms of the state equation coefficients.

### 2.6.1. Introduction

The minimal state representation of an  $N^{\text{th}}$  order block filter as in Fig. 2-1 can be written as

$$R_{n+1} = AR_n + BX_n \quad (2-21a)$$

$$Y_n = CR_n + DX_n \quad (2-21b)$$

where  $A$ ,  $B$ ,  $C$  and  $D$  are, respectively,  $N \times N$ ,  $N \times L$ ,  $L \times N$  and  $L \times L$  constant matrices.  $R_n$ ,  $X_n$  and  $Y_n$  are, respectively, the state vector, the input vector and the output vector at time  $n$ , where  $X_n$  and  $Y_n$  are defined as in equations (2-3). This representation can be uniquely characterized by the constant matrices ( $A, B, C, D$ ). The block diagram of this block filter is shown in Figure 2-10.

Actually, the state matrix  $A$  can be of any size greater than  $N$ , if introducing some dummy states (poles). Then the sizes of matrices  $B$  and  $C$  should change accordingly. This size change is acceptable because the only thing that matters for digital filters is the zero-state response, or equivalently, the transfer function. This state equation with larger matrices will give us the same transfer function after the pole and zero cancellation and hence the realization is not minimal. For instance, the direct form implementation of a  $2^{\text{nd}}$  order filter as shown in Figure 2-7a with block size 1 can be expressed by state equations with the following matrices

$$A = \begin{bmatrix} b_1 & b_2 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



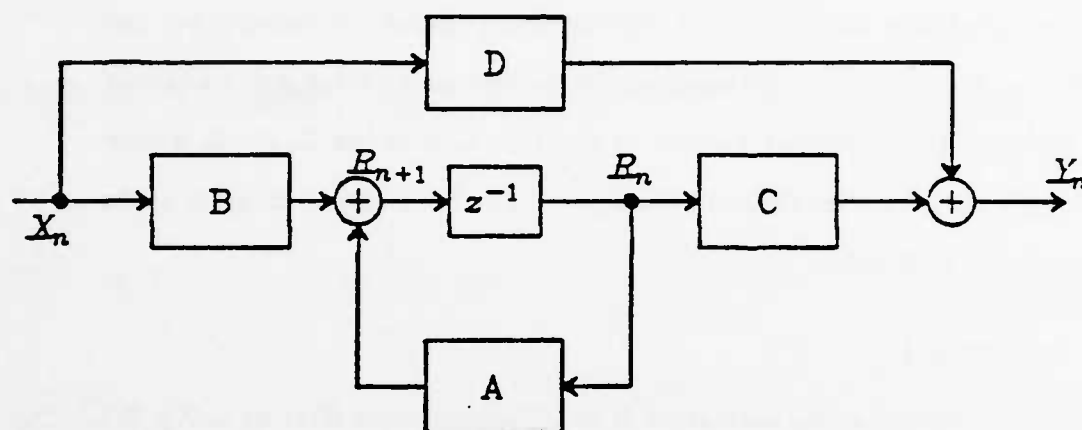


Figure 2-10 Block Diagram of Block State Filters

$$C = [a_0, a_1, a_2] \quad D = 0 \quad (2-22)$$

We will concentrate here on the minimal realization only.

The state matrix  $A$  plays a very important role in achieving high speed filters, since it is the only feedback matrix in the state equations. This state matrix can vary from a full matrix to a very simple one such as the Jordan form. The diagonality of the Jordan form is extremely important in simplifying the filter structure, since the operation of each entry on the diagonal can be performed independently. Thus, the computation path on the feedback term can be divided to several parallel paths with much less computation on each path. Since the execution time on the feedback term decides the filter speed, we can easily trade hardware with speed for a diagonal state matrix.

A diagonal state matrix usually introduces complex numbers to its entries, since the diagonal elements are equivalent to the filter poles in this case. Thees

complex numbers unnecessarily increase the computation because all the filter coefficients as well as the input data are real numbers. All the states can also be real numbers if choosing the correct states. Combining each complex conjugate pair of poles, a complex  $2 \times 2$  diagonal submatrix can be converted into a real  $2 \times 2$  full matrix. This will transform the state matrix from a diagonal Jordan form into a block diagonal form. For each state, a complex multiplication is transformed into two real multiplications. Hence, the computational rate is reduced to one half.

Although these equations exist in theory, they are not easy to obtain directly from a filter difference equation, which is the form we usually have for implementation. Fortunately, computing one out of every  $L$  state vectors, the state equations can be easily obtained from the simple state equations, which are, in turn, fairly easy to obtain from the scalar difference equation. If synthesized from the second and/or first order filters, the state matrix will be block diagonal. Thus, a step-by-step procedure to obtain the block diagonal state equations from the SISO filter design will be shown in the next subsection.

However, the four matrices for a given filter transfer function are not unique. Actually, there are infinite number of matrices which will give rise to the same transfer function by choosing different states. Even if restricted to the minimal realization, as will see later, there is still a lot of freedom to choose the coefficients. Filters, which have a block diagonal state matrix and low roundoff noise, will be derived in Chapter 5. In this section, synthesizing a block state filter from scalar second order filters will be shown. The realization of a low noise second order filter will also be given in Chapter 5.

### 2.6.2. Single Input Single Output Filter

### 2.6.2.1. High Order Filters

The minimal SISO state space representation of an  $N^{\text{th}}$  order filter can be written as

$$r_{n+1} = Ar_n + bx_n \quad (2-23a)$$

$$y_n = cr_n + dx_n \quad (2-23b)$$

where  $x_n$  and  $y_n$  are scalars rather than vectors. Taking the  $z$  transform on equations (2-23), we get

$$zR(z) = AR(z) + bX(z) \quad (2-24)$$

$$Y(z) = cR(z) + dX(z) \quad (2-25)$$

Substituting the  $R(z)$  of (2-24) into (2-25), we get

$$H(z) = \frac{Y(z)}{X(z)} = c(zI - A)^{-1}b + d \quad (2-26)$$

For a given filter transfer function, if all the poles are distinct, one can do partial fractional expansion on the original filter such that the transfer function is a summation of several second and/or first order filters. For the case of all distinct complex poles, this transfer function can be written as

$$H(z) = d + \sum_{i=1}^P \frac{\alpha_i^1 + \alpha_i^2 z^{-1}}{1 - \beta_i^1 z^{-1} - \beta_i^2 z^{-2}} \quad (2-27)$$

where  $P$  is the number of poles in the upper half  $z$ -plane. Each term in the summation can be realized in the state space. Suppose the state space realization of the  $i^{\text{th}}$  term is characterized by  $(A^i, b^i, c^i, 0)$ , where  $A^i$  is a  $2 \times 2$  matrix. These three matrices can be related to equation (2-27) by

$$c^i(zI_2 - A^i)^{-1}b^i = \frac{\alpha_i^1 + \alpha_i^2 z^{-1}}{1 - \beta_i^1 z^{-1} - \beta_i^2 z^{-2}} \quad (2-28)$$

where  $I_2$  is a  $2 \times 2$  identity matrix. Then, it is straightforward to obtain the high order filter  $(A, b, c, d)$  by defining the filter coefficients as

$$A = \begin{bmatrix} A^1 & 0 & \dots & 0 \\ 0 & A^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & A^P \end{bmatrix} \quad b = \begin{bmatrix} b^1 \\ b^2 \\ \vdots \\ b^P \end{bmatrix}$$

$$c = [c^1, c^2, \dots, c^P] \quad (2-29)$$

This can be easily verified by plugging these coefficients into equation (2-26) and then comparing with equations (2-27) and (2-28).

$$\begin{aligned} H(z) &= c(zI - A)^{-1}b + d \\ &= [c^1, c^2, \dots, c^P] \begin{bmatrix} (zI_2 - A^1)^{-1} & 0 & \cdot & 0 \\ 0 & (zI_2 - A^2)^{-1} & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & (zI_2 - A^P)^{-1} \end{bmatrix} \begin{bmatrix} b^1 \\ b^2 \\ \cdot \\ b^P \end{bmatrix} + d \\ &= [c^1(zI_2 - A^1)^{-1}, c^2(zI_2 - A^2)^{-1}, \dots, c^P(zI_2 - A^P)^{-1}] \begin{bmatrix} b^1 \\ b^2 \\ \cdot \\ b^P \end{bmatrix} + d \\ &= d + \sum_{i=1}^P c^i (zI_2 - A^i)^{-1} b^i \end{aligned}$$

This is exactly the same as equation (2-27) if replacing all the matrices  $A^i$ ,  $b^i$  and  $c^i$  by (2-28).

It is clear that the matrix  $A$  is block diagonal and each submatrix is of size  $2 \times 2$ . Therefore, from equation (2-29), a high order filter can be synthesized, once a realization of a second order filter in state space is developed. A minimal second order filter design will be discussed in the next subsection.

For the filter with higher multiplicity poles, equation (2-27) is no longer valid. Some terms in the summation must have a denominator with degree of higher than 2. This in turn will affect the structure of matrix  $A$ .  $A$  can stay in a block diagonal form, if factoring the original transfer function into several product terms as in equation (2-10). Then, each term can be independently realized with a block diagonal state matrix. The filter now becomes a cascade connection of block filters with simple poles only.

### 2.6.2.2. Second Order Filter Design

Suppose the  $i^{th}$  term in the summation of equation (2-27) is of the following form.

$$H(z) = \frac{r}{z-p} + \frac{r^*}{z-p^*} \quad (2-30)$$

A filter in state space design can be readily obtained by defining the coefficients as

$$\begin{aligned} A &= \begin{bmatrix} p & 0 \\ 0 & p^* \end{bmatrix} & b &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \\ c &= [c_1, c_2] \end{aligned} \quad (2-31)$$

where  $b_1 c_1 = r$  and  $b_2 c_2 = r^*$ . This can be easily verified by substituting the coefficients in (2-31) into (2-26) and comparing with (2-30). Actually, depending on how we choose the states, these equations can result in infinite number of realizations with different coefficients. One useful example would be making all coefficients real numbers.

Real matrices would be desirable, since the computational rate is lower. For a complex state matrix, two complex multiplications or equivalently, 8 real multiplications are needed, while in the real state matrix case, only 4 multiplications are required. Theoretically, a real state equation is obtainable, since the filter itself is real.

Before showing how this can be achieved, let us state a useful theorem as follows

#### *Theorem 2-1 - Equivalent Realization*

Given a realization (A, b, c, d) for an  $N^{th}$  order filter, define  $\bar{r} = T\bar{r}$ , where T is an  $N \times N$  nonsingular matrix. Then  $(TAT^{-1}, Tb, cT^{-1}, d)$  realizes the same filter and the state equations become

$$\begin{aligned} \bar{r}_{n+1} &= TAT^{-1}\bar{r}_n + Tb\bar{x}_n \\ y_n &= cT^{-1}\bar{r}_n + d\bar{x}_n \end{aligned}$$

If set the nonsingular matrix  $T$  to be

$$\frac{1}{2} \begin{bmatrix} 1 & 1 \\ j & -j \end{bmatrix}$$

the transformed system of (2-31) can be real, if  $b_1$ ,  $b_2$  and  $c_1$ ,  $c_2$  are complex conjugate pairs. The new set of coefficients will be

$$\begin{aligned} A &= \begin{bmatrix} \operatorname{Re}(p) & \operatorname{Im}(p) \\ -\operatorname{Im}(p) & \operatorname{Re}(p) \end{bmatrix} & b &= \begin{bmatrix} \operatorname{Re}(b_1) \\ -\operatorname{Im}(b_1) \end{bmatrix} \\ c &= 2 \begin{bmatrix} \operatorname{Re}(c_1) & \operatorname{Im}(c_1) \end{bmatrix} & d &= d \end{aligned}$$

Obviously, the number of nonsingular matrices  $T$  is infinite; hence, there are infinite ways to choose matrices  $A$ ,  $b$  and  $c$ . A low roundoff noise filter can be achieved without increasing computation by choosing an optimal matrix  $T$ . Jackson et al., [17] derived an optimal second order filter in state space design, and Barnes [18, 19, 20, 21] showed another realization called the normal filter which has low noise and is free of autonomous overflow limit cycles. These two realizations will be derived in Chapter 5.

### 2.6.3. Multi-Input Multi-Output Filter

In single-input single-output filter design, the partial fractional expansion of the original transfer function is usually obtained before realizing the filter in state space. However, for a block filter the transfer function is in a matrix form and usually not easy to obtain. Barnes and Shinnaka [22] derived part of the transfer function, from which the difficulty in obtaining the general solution for the transfer function is evident. Fortunately, it is not necessary to know the transfer function to implement the filter. The filter can be implemented from the state equations directly, which are fairly easy to obtain.

#### 2.6.3.1. Formulation

Suppose we are given a SISO representation  $(\hat{A}, b, c, d)$ . Relating the state vectors of MIMO and SISO as  $R_n = r_{nL}$ , a representation  $(A, B, C, D)$  in the MIMO

system can be easily obtained, where the matrices are defined as follows

$$\begin{aligned}
 A &= \bar{A}^L & B &= [\bar{A}^{L-1}b \quad \bar{A}^{L-2}b \quad \dots \quad b] \\
 C &= \begin{bmatrix} c \\ c\bar{A} \\ \vdots \\ c\bar{A}^{L-1} \end{bmatrix} & D_{ij} &= \begin{cases} 0, & i < j \\ d, & i = j \\ c\bar{A}^{i-j-1}b, & i > j \end{cases}
 \end{aligned} \tag{2-32}$$

The state equations can be induced easily as follows

$$\begin{aligned}
 R_{n+1} &= r_{(n+1)L} \\
 &= \bar{A}r_{(n+1)L-1} + bx_{(n+1)L-1} \\
 &= \bar{A}[\bar{A}r_{(n+1)L-2} + bx_{(n+1)L-2}] + bx_{(n+1)L-1} \\
 &= \bar{A}^2r_{(n+1)L-2} + \bar{A}bx_{(n+1)L-2} + bx_{(n+1)L-1} \\
 &\dots\dots \\
 &= \bar{A}^Lr_{nL} + \bar{A}^{L-1}bx_{nL} + \dots + \bar{A}bx_{(n+1)L-2} + bx_{(n+1)L-1} \\
 &= \bar{A}^L R_n + \begin{bmatrix} \bar{A}^{L-1}b, \bar{A}^{L-2}b, \dots, b \end{bmatrix} \begin{bmatrix} x_{nL} \\ x_{nL+1} \\ \vdots \\ x_{(n+1)L-1} \end{bmatrix} \\
 &= AR_n + BX_n
 \end{aligned} \tag{2-33}$$

For the output equation, each output sample in one block can be calculated as

$$\begin{aligned}
 y_{nL} &= cr_{nL} + dx_{nL} = cR_n + dx_{nL} \\
 y_{nL+1} &= cr_{nL+1} + bx_{nL+1} = c\bar{A}r_{nL} + cbx_{nL} + bx_{nL+1} \\
 &= c\bar{A}R_n + cbx_{nL} + bx_{nL+1} \\
 &\dots\dots \\
 y_{(n+1)L-1} &= c\bar{A}^{L-1}R_n + c\bar{A}^{L-2}x_{nL} + \dots + cbx_{(n+1)L-2} + dx_{(n+1)L-1}
 \end{aligned}$$

Expressing the above equations in a matrix form, we get

$$Y_n = CR_n + DX_n \tag{2-34}$$



### 2.6.3.2. Some Important Properties

Barnes[22] proved some theorems about block state representation, some of which are important to us and are stated in this section. In the following discussion, denote a state-space realization of the scalar system by  $(\tilde{A}, b, c, d)$ , and its corresponding MIMO system related by equation (2-32), is denoted by  $(A, B, C, D)$ . We also denote the mapping between these two systems by  $(\tilde{A}, b, c, d) \rightarrow (A, B, C, D)$ .

*Theorem 2-2. Irreducibility Invariance:*

$(A, B, C, D)$  is an irreducible realization iff  $(\tilde{A}, b, c, d)$  is an irreducible realization.

*Theorem 2-3. Equivalence Invariance:*

$(TAT^{-1}, Tb, cT^{-1}, d) \rightarrow (TAT^{-1}, TB, CT^{-1}, D)$  iff  $(\tilde{A}, b, c, d) \rightarrow (A, B, C, D)$ , where  $T$  is any  $N \times N$  invertible matrix.

*Theorem 2-4. Completeness:*

If  $(A, B, C, D)$  is an irreducible realization of a block-shift invariant MIMO system, then there exists a unique scalar system realization,  $(\tilde{A}, b, c, d)$  such that  $(\tilde{A}, b, c, d) \rightarrow (A, B, C, D)$ .

### 2.6.4. Block Cascade and Parallel Forms

In scalar input scalar output filter design, the most popular design approach is to decompose the original filter equation into a cascade or a parallel connection of second order filters. This idea can also be applied to the block filter design. All the second order sections can be processed in a parallel or a pipelined fashion. The block state filter design then reduces to a block second order filter design. For a second order filter, the state matrix  $A$  is always of size  $2 \times 2$  which greatly simplifies the filter design process.

Zeman and Lindgren[23] proposed a inner product arithmetic unit which can compute the matrix-vector multiplications in the block state equations.

They also combine the state and output vectors into one larger vector and the equation of a second order filter becomes

$$\begin{bmatrix} R_{n+1} \\ Y_n \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} R_n \\ X_n \end{bmatrix} \quad (2-35)$$

where A, B, C and D are, respectively,  $2 \times 2$ ,  $L \times 2$ ,  $2 \times L$  and  $L \times L$  constant matrices. Then, the next state and current output are obtained by an inner product hardware circuit. For each second order filter, there are  $2+L$  inner products, each one with size  $2+L$ .

## 2.7. Conclusions

Several parallel algorithms for an IIR filter realization have been shown in this chapter. For the SISO filters, the cascade and parallel forms are well-known structures and easy to realize. The systolic array approach is a very efficient structure due to its two way pipelining of the input and output sequences. It can also be easily expanded to any order by appending or deleting cells at the end. Barnwell's filter is of SIMD mode and hence very easy to realize. Only one set of program is needed to realize the whole filter. The advantage of the latter two structure is that they can be directly realized from the difference equation and no factoring or partial fractional expansion is required. On the other hand, the roundoff noise behavior is equivalent to that of the direct form implementation, which would be a serious problem.

For block filters, the I/O and state space formulations are shown. Block filters are extremely suitable for VLSI processing due to their increased parallelism over SISO filters. A very efficient structure in block state space design will be shown in the next chapter.

## CHAPTER 3

### A HIGH SPEED FILTER STRUCTURE

An extremely efficient structure for the block filter design will be shown in this chapter. This structure can achieve any desired speed at the expense of increased overall filter latency and hardware. The latency is defined as the time for a sample to pass through the filter from the input to the output. Furthermore, this high speed can be achieved with a structure which requires only localized communication. This structure is realized with the technique of block processing described in Chapter 2. As will see later, block state filters have a more efficient structure than the I/O filters; hence, a detailed discussion of the state filters will be given in this chapter.

For the implementation of block filters, the filter equation alone does not specify the operation assignment and the interconnection among PE's, whereas for the systolic or SSMD filter design cases, the filter equation completely specify the actual function of each PE. For the systolic approach, even the structure is completely defined by the filter equation, whereas for Barnwell's filter, once the processor number is given along with the equation, the actual interconnection is immediately available. However, given a filter equation in a block form, there is still a lot of freedom that we have to implement the filter. The ultimate goal is to find a structure which can sustain high throughput rate while keeping the interprocessor communication localized.

#### 3.1. FIR Filter Design

FIR filters are easy to be pipelined or to be put in parallel to achieve very high speed. Both pipelining and parallel forms can be easily extended to any size. Consequently, the overall speed can also be easily increased to any value.

We will show in this section how the speed of an FIR filter can be increased by utilizing concurrency. The parallel realization of IIR filters will be discussed in the following sections.

### 3.1.1. Formulation

Since the feedforward computation of the three matrices B, C and D in the block state space design is very similar to the FIR filter operation, knowledge about the FIR filter implementation would help us design the block state filters. As mentioned before, the implementation of an FIR filter is equivalent to the realization of a finite convolution of two sequences. The finite convolution is a scalar operation, which can be represented by the following equation

$$y_n = \sum_{k=0}^M a_k x_{n-k} \quad (3-1)$$

This is equivalent to setting N to zero in equation (2-1).

As mentioned earlier, the FIR filters can also be represented as an MIMO system which has the following equation

$$Y_n = AX_n + BX_{n-1} \quad (3-2)$$

where A and B are equivalent to the matrices  $A_0$  and  $A_1$  in (2-17) respectively. The input and output vectors,  $X_n$  and  $Y_n$ , are defined as in (2-3). Equation (3-2) is true only if the block length L is not smaller than the filter order M. Otherwise, more matrix-vector multiplying terms will be summed up in (3-2). Actually, when L reduces to 1, equation (3-2) becomes a scalar equation as (3-1).

Combining  $X_n$  and  $X_{n-1}$  into one large vector, (3-2) can be transformed into an equation, which consists of a single matrix-vector multiplication.

$$Y_n = \begin{bmatrix} B & A \end{bmatrix} \begin{bmatrix} X_{n-1} \\ X_n \end{bmatrix} \quad (3-3)$$

Since the left L-M columns of matrix B are all null vectors (See  $A_1$  in (2-17)), the composite matrix can be reduced to another matrix  $A'$  of size L by L+M, and

only the last  $M$  samples in  $X_{n-1}$  are needed. If define the new composite vector as  $X_n$ , (3-3) can be written as

$$Y_n = A'X_n$$

From the definition of matrices  $B$  and  $A$ , it is clear that matrix  $A'$  is a banded matrix. This matrix is shown in (3-4) for the case of  $M=3$  and  $L=4$ .

$$\begin{bmatrix} Y_{n1} \\ Y_{n2} \\ Y_{n3} \\ Y_{n4} \end{bmatrix} = \begin{bmatrix} a_3 & a_2 & a_1 & a_0 & 0 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ 0 & 0 & a_3 & a_2 & a_1 & a_0 & 0 \\ 0 & 0 & 0 & a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} X_{n1} \\ X_{n2} \\ X_{n3} \\ X_{n4} \\ X_{n5} \\ X_{n6} \\ X_{n7} \end{bmatrix} \quad (3-4)$$

Since the first three entire in  $X_n$  are in  $X_{n-1}$  and the others are in  $X_n$ , it is obvious that the first three samples are equal to the last three samples in the previous cycle.

### 3.1.2. Implementations

The direct form implementation of equation (3-1), which consists of several delay lines, multipliers and a large adder, is shown in Figure 3-1. The throughput of this structure is fixed for a given chip speed. The only way to increase the throughput, as suggested in Figure 1-2, is to duplicate this section so that a number of sections can simultaneously work on successive output samples. The accompanied problem is the extremely complex data flow pattern from the data source to all the sections. A even more serious problem would be the inefficiency of the hardware usage. With a single section, the output sequence can keep flowing out of the structure by utilizing the stored data on the delay lines. Thus, each input sample is used  $M+1$  times, while staying on the delay line, to generate  $M+1$  different output samples. If more sections are duplicated, the time gap between two adjacent output samples from each section is longer. Hence, each input sample is used less often. This inefficiency implies

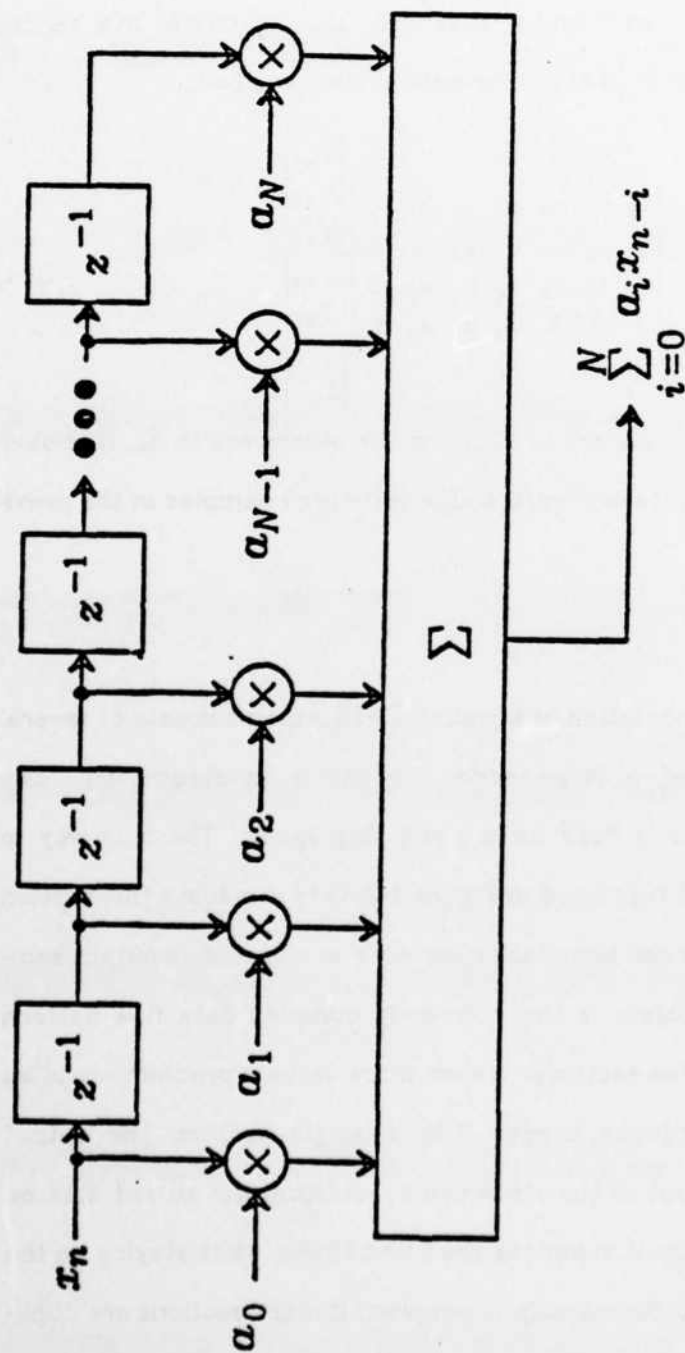


Figure 3-1 Direct Form Implementation of FIR Filters

that more internal memory space is required and each input sample has to be duplicated for all the affected sections.

Kung[8] devised several systolic arrays for implementing a finite convolution, which can be used to realize an FIR filter. One of these arrays is drawn in Figure 3-2.

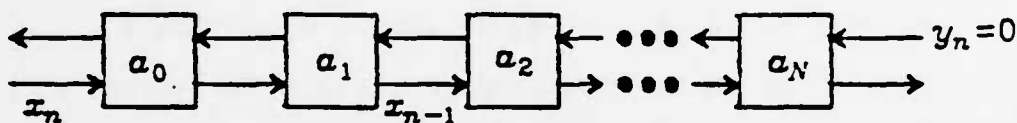


Figure 3-2 Linear Systolic Array for the Finite Convolution

The coefficients stay in the cells, whereas the input and output sequences are traveling in opposite directions at the same speed. Consecutive  $x_i$ 's and  $y_i$ 's are separated by two clock cycles to ensure that each  $x_i$  is able to meet every  $y_i$ . Obviously, this array is capable of outputting a  $y_i$  every two cycles; hence, only one-half the cells work at any given time. This structure, although very simple, also suffers from the problems of limited speed and inefficiency if more sections are duplicated.

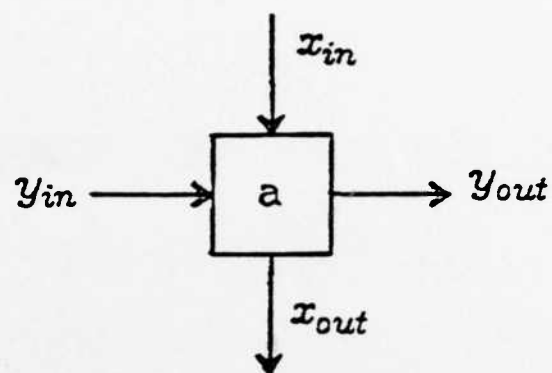
In contrast to SIS0 processing, block processing can easily increase the throughput rate by increasing the size of vector  $X_n$ . Although Kung's linear array for matrix-vector multiplication[6] is more efficient for a banded matrix, the limitation on speed is still a problem. If putting more such arrays in parallel, the data distribution is still very complex. A very efficient structure for implementing (3-2) or (3-3) will be shown in this section. This structure has a very regular interconnection, and hence it can be easily expanded without complicating the communication environment. Furthermore, all the cells can still



be efficiently exploited after the structure is expanded. This structure utilizes the idea of the systolic array but with a little modification. In order to process the input vectors continuously, a two-dimensional array rather than a linear one is used.

Figure 3-3a shows the basic function of one cell, and 3-3b shows the internal structure of such cell. The two latches in Figure 3-3b ensure that all the data transfer can be done synchronously. The interconnection of such cells to implement (3-4) is illustrated in Figure 3-4. The two dimensional array in Figure 3-4 accepts input vectors from the top and outputs vectors to its right. The cells on the  $i^{\text{th}}$  row calculate the  $i^{\text{th}}$  output sample in the vector  $Y_n$ . On the vertical transmission, no computation is involved; hence, a short bus can be used to send data to all the cells on the same column. After finishing its job, each cell sends its updated output to the right cell. The rightmost cell sends out the final result calculated by the row that this cell is located.

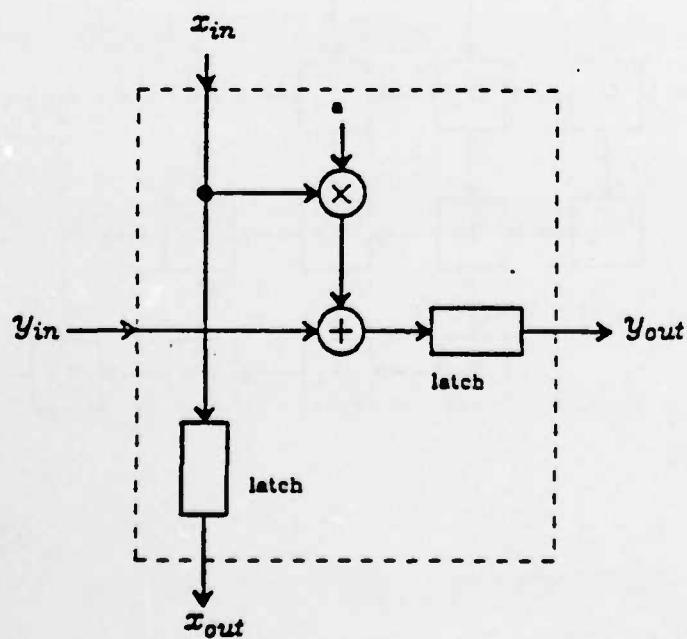
In Figure 3-4, each row is equivalent to the realization of an SISO FIR filter. All the rows work simultaneously at the same speed but with a fixed time skew on the starting time. If the two latches are controlled by a single clock, this time skew equals the execution time of one cell. Thus, the number of rows is equivalent to the number of FIR filters working independently. Obviously, the overall throughput rate depends on the number of rows in the array. If neglecting the data transmission overhead, the throughput rate is  $L$  times higher than a single FIR filter for a vector of size  $L$ . Therefore, the throughput rate can be easily increased by simply appending more rows to this array. There is no limit on the number of rows that can be appended; hence, this structure can achieve any arbitrary speed with a simple expansion. The only thing that has to be taken care of is the data distributor which distributes the input data sequence to the right column. Since some samples are used in two adjacent vectors at



$$x_{out} \leftarrow x_{in}$$

$$y_{out} \leftarrow y_{in} + a \cdot x_{in}$$

(a)



(b)

Figure 3-3: PE Structure for Block FIR Filter Realization (a) Function of One PE (b) Internal Structure of One PE

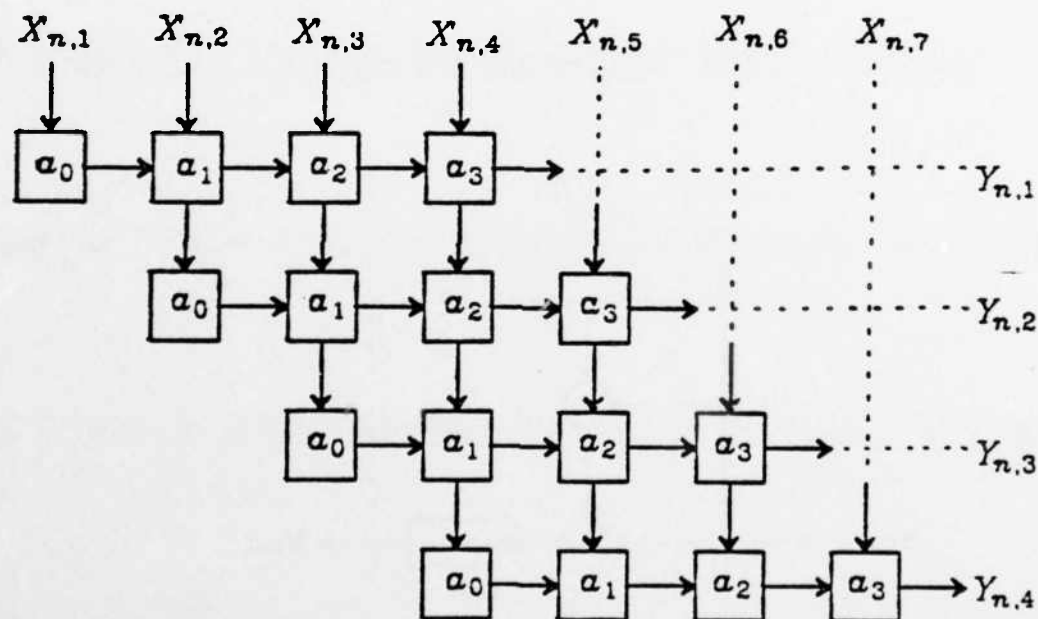


Figure 3-4 Two Dimensional Systolic Array for Block FIR Filter Implementation

different clock cycles, care must be taken in the data distributor design to reflect the actual data flow. However, this data distributor is much simpler than that in Figure 1-2. This is true due to the fact that each input sample goes through a fixed path whereas in Figure 1-2, each sample has to be sent to  $M+1$  different sections for the computation of  $M+1$  output samples.

If a bus is used for the vertical transmission on each column, the starting time of two adjacent rows are separated by the input sampling period rather than by the execution time of each cell. The latter is usually much longer than the former. If this array functions synchronously, a latch must be inserted between any two adjacent cells on the same column as well as the places corresponding to the upper right zero's in matrix  $A'$ . The output samples will then be skewed by the execution time of one cell. Thus, although the throughput stays unchanged, the response time, or filter latency, increases.

### 3.2. Real Time Constraint for the IIR Filter Design

Conventional wisdom would say that unlike the FIR filter, for a given speed of hardware the sampling rate of the IIR filter is limited by the feedback inherent in the recursion. It will now be shown that this is not the case, and specifically that an architecture with unlimited speed (similar to the FIR filter) can be defined. This architecture of course exploits parallelism, and can be implemented in a fashion which requires only local communication. The representation we will describe is based on a block state realization of the IIR filter.

Block filters, whether I/O or state space, consist of several matrix-vector multiplications. From Figures 2-8 and 2-9, it is clear that among these multiplications, only one term in each graph operates recursively. Since this feedback operation can not be pipelined and has to finish in one block period, the overall throughput rate depends solely on the implementation of this feedback term.

The block period is defined as the time needed to accumulate all the samples in the buffer. Hence, this period is  $LT_s$ , where  $L$  is the buffer size and  $T_s$  is the scalar input sampling period. As for the other matrices, the only constraint is that the operation time for each input vector can not exceed the block period. An arbitrary number of delays can be inserted on the path computing these feedforward matrices without affecting the overall throughput rate. Hence, the parallel implementation of the feedback matrix will be discussed first and the implementation of the other matrices will be treated afterwards.

For a minimal realization of a block state filter, the size of the feedback state matrix equals the filter order  $N$ , which is fixed for a given filter independent of block size  $L$ . Certainly, the value of each entry does change with the block size; however, the number of entries stays the same. Therefore, the execution time of this matrix depends only on the filter order and the processor speed but not on the block size, if a single processor is used for computing this feedback matrix. Suppose the time to perform a multiplication and an addition is  $t_m$  and  $t_a$  respectively, and the input sampling period is  $T_s$ . The execution time of a full state matrix is  $N^2[t_m + t_a]$ , where  $N$  is the order of the filter. The minimum block size that guarantees real time is the smallest integer which satisfies the following condition

$$L \geq \frac{N^2(t_m + t_a)}{T_s} \quad (3-5)$$

This equation guarantees that the execution time of the feedback matrix multiplying a state vector is less than a block period. Obviously, it is always possible to find a finite number  $L$  satisfying the above condition, no matter how fast the input sampling rate is and how slow the processor speed is. In other words, the state space design of IIR filters can achieve a speed as high as we want without any limit. Another way to describe this real time processing is that a vector of samples rather than a scalar can be generated in a fixed time, which is

$N^2(t_a + t_m)$ . Thus, the vector throughput rate, which is defined as the processing speed of one vector, is fixed. The scalar throughput rate, which is  $L$  times that of the vector through rate, can be easily increased to any value by increasing the block size.

Although any desired speed can be achieved according to equation (3-5), the large value of  $L$  might restrict the applicability of this filter structure, since a large  $L$  implies a long filter latency, as well as higher computational rate. A higher computational rate implies more hardware required. The size of the hardware may look unrealistic when  $L$  is very large. Furthermore, no real time signal processing systems can tolerate infinite delay. Hence, if the overall latency is intolerable, higher speed chips are required to reduce the block size and hence the filter latency. Since  $L$  is directly related to the filter latency, it is always beneficial to reduce this number while keeping the original speed.

For the block diagonal state matrix case, each submatrix on the diagonal is a  $2 \times 2$  matrix and is decoupled from any other submatrices. Therefore, all the submatrices can be processed in parallel without communicating with one another. Hence, the execution time of the state matrix reduces to that of a  $2 \times 2$  matrix. This execution time, which depends only on the chip speed but not on the filter order, equals  $4(t_m + t_a)$ . Equation (3-5) can then be modified as follows

$$L \geq \frac{4(t_m + t_a)}{T_s} \quad (3-6)$$

For a high order filter, the value  $L$  can be greatly reduced compared to equation (3-5). For a filter with complex poles only, all the submatrices have exactly the same execution time; hence, no idle time is present in any cell similar to the pipelining of second order filters. The overall speed of a high order filter then depends on that of a second order section and not on the filter order.

In the following sections, we will discuss the implementation of a block filter with block diagonal state matrix only. Even though the block size and the

implementation of the state matrix are fixed, good implementations of the other three matrices B, C and D deserve further investigation. We will show how an implementation with only localized communication can be achieved.

### 3.3. Direct Implementation

A naive way to implement the three matrix-vector multiplications is to treat each of them as several inner products and then transmit the outcomes to the appropriate places. Hence, a tree structure inner product structure as shown in Figure 3-5 is a natural selection. Each cell in this structure contains either a multiplication or an addition. Since the execution time of each cell is much shorter than that of one submatrix in A, several cells can be combined into one processor. However, it is very unlikely that we can have an assignment such that all the cells, including those working on the feedback matrix, have exactly the same execution time. Hence, waiting time for some cells is unavoidable, and this results in requiring more hardware.

Another problem arising from this structure is the complexity of interprocessor communication. First, the input samples  $x_i$  are sent to all rows of matrices B and D. This requires a broadcasting type of communication link between the data source and the matrices operating on the input sequence. Second, the output data from the state matrix has to be sent to matrix C. If implementing C as inner products, send each output sample from matrix A is sent to every inner product structure. This also results in a broadcasting type of transmission.

This rather complex communication pattern usually requires some dedicated chips to perform the data switching. These extra switching processors along with the inevitable processor idle time make this direct implementation very undesirable. Furthermore, the complex switching network might cause a serious delay for each message transmission. Although this delay does not



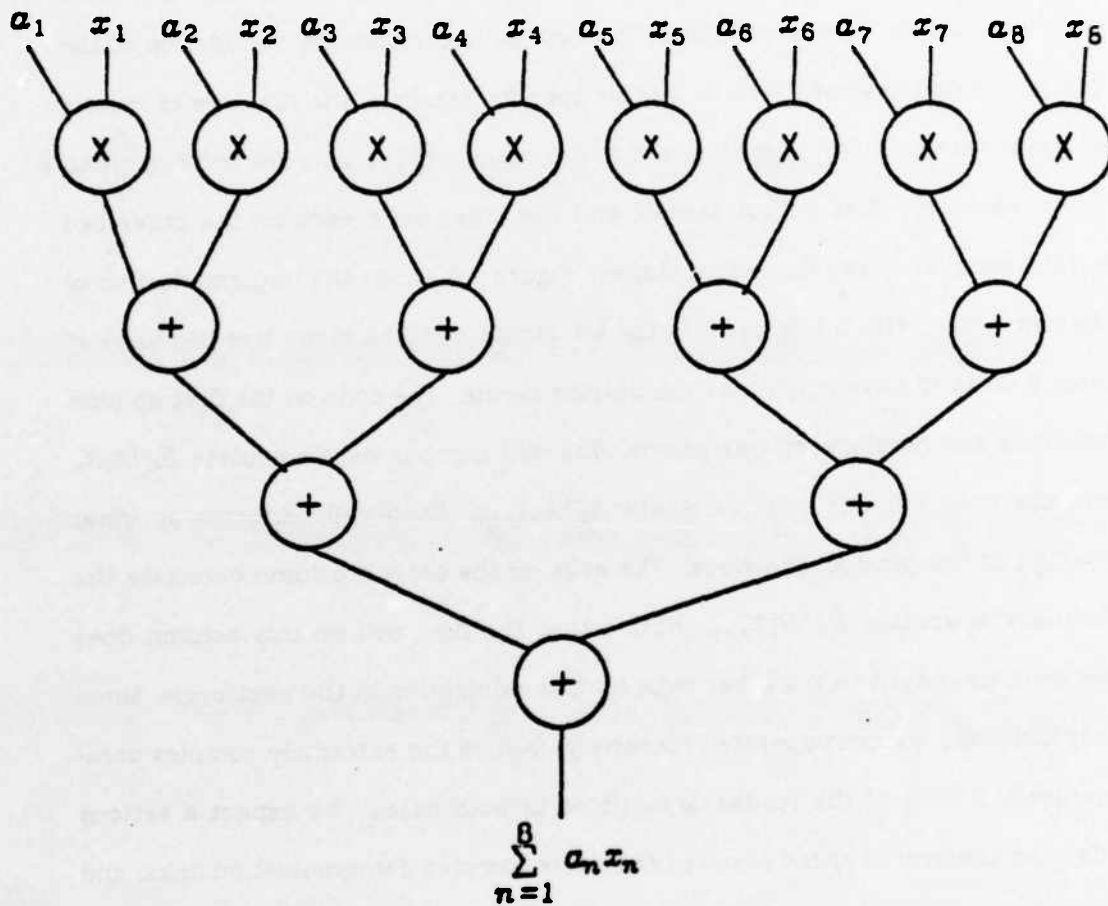


Figure 3-5 Tree Structure for Realizing Inner Product

affect the overall throughput rate, since the delay is on the feedforward path, it would surely increase the overall filter latency.

Figure 3-6 shows the interconnection of PE's for a six order filter with simple complex poles only. Cells 1 to 6 calculate  $BX_n$ , with each row of matrix B assigned to one cell. Cells 7, 8 and 9 perform the state feedback operation with each one containing a  $2 \times 2$  matrix. The rest cells perform the calculation of the composite matrices of C and D. In this specific example, the first row of matrix contains three 1's and three 0's, and hence no multiplications are involved. Cell 10 works on the first output sample and the other cells work on the other two output samples in two cascading stages. Figure 3-7 shows the implementation of the same filter with block size 7 in the I/O structure. The block size has to be at least 8 so as to have only three multiplying terms. The cells on the first column calculate the feedforward operations. The odd number cells calculate  $B_0^{-1}A_0X_n$  and the even number cells calculate  $B_0^{-1}A_1X_{n-1}$ . Each cell performs an inner product of the product matrices. The cells on the second column calculate the feedback operation  $B_0^{-1}B_1Y_{n-1}$ . Notice that the first cell on this column does not send its output to the other cells for the calculation in the next cycle, since only the lower six cells operate recursively. Notice the extremely complex communication links of the feedback matrices in both cases. We expect a serious effect on the overall speed resulted from this complex communication links, and will discuss it later.

### 3.4. Systolic Arrays

As mentioned in section 2-3, systolic arrays make localized communication possible while attaining high throughput. If we can utilize the systolic idea for realizing the operations of matrices B, C and D, the overall structure will be regular and have only localized communication. Since global communication is very expensive in VLSI whether it is on chip or off chip, the localized

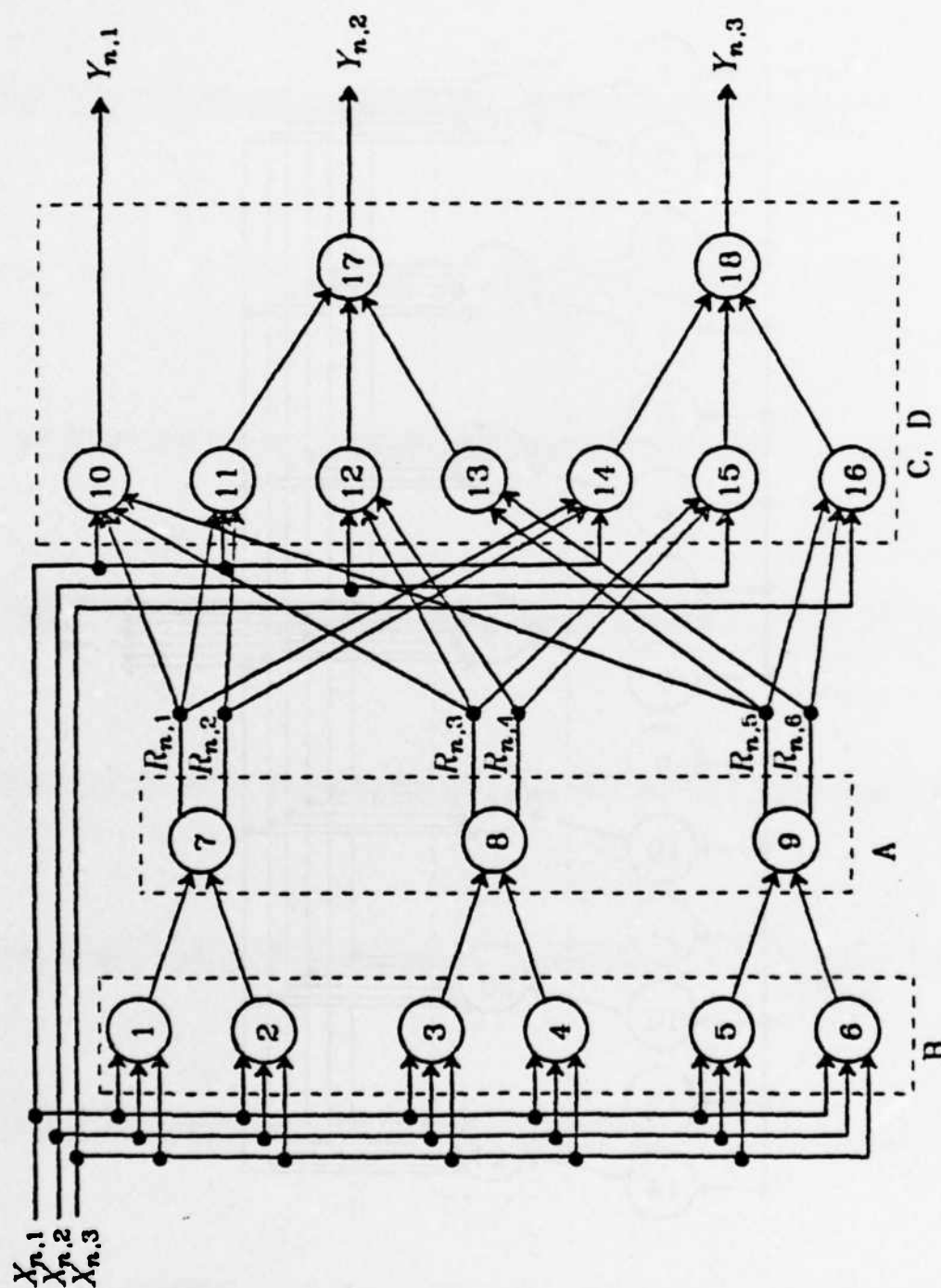


Figure 3-8 A 6th Order Block State Filter Structure

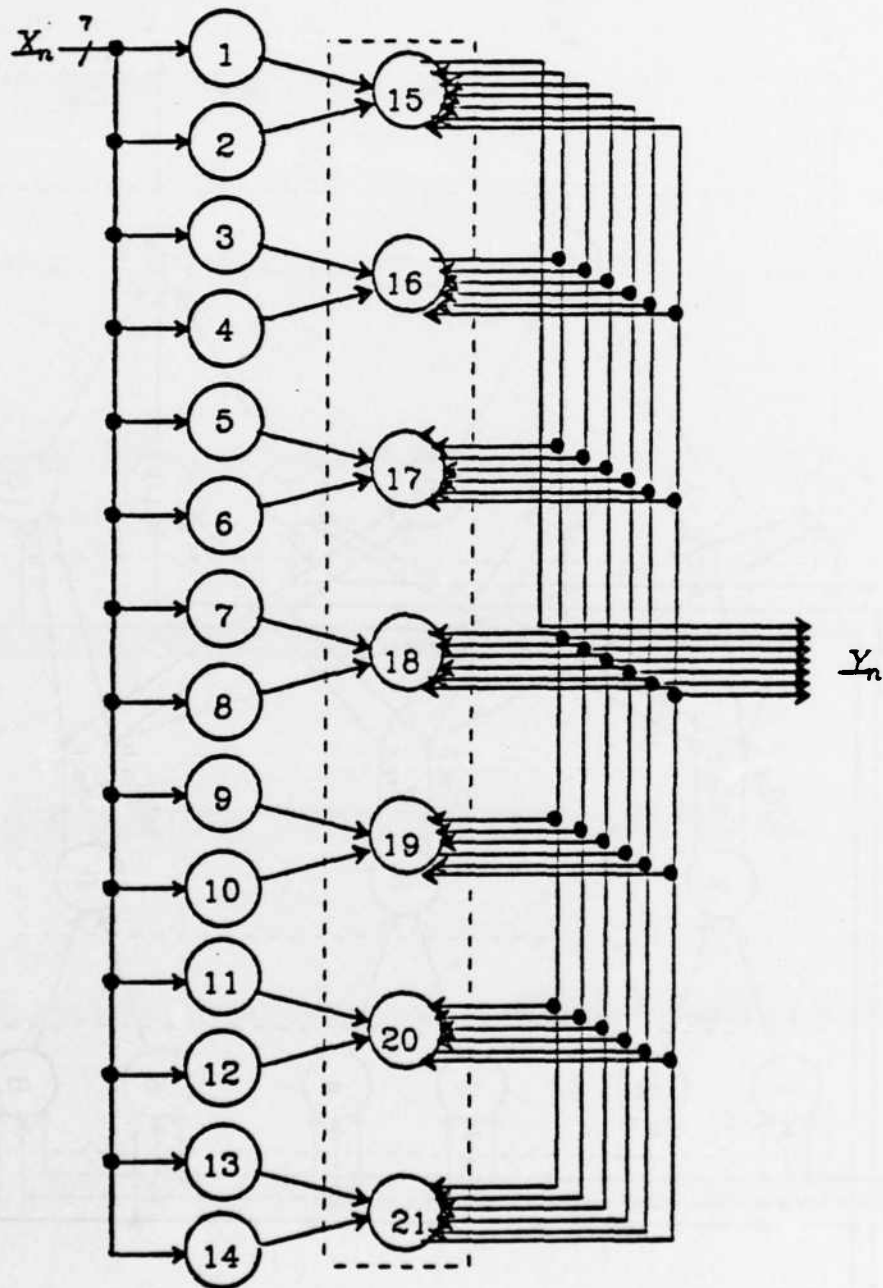


Figure 3-7 A 6<sup>th</sup> Order Block I/O Filter Structure

communication makes the systolic array very attractive in VLSI applications.

To implement the matrix-vector multiplication, Kung[6] showed a linear systolic array which can be applied to the implementation of matrices B, C and D. However, he assumed that this multiplication is done only once, while in digital filter implementation, the same matrix multiplies essentially an infinite number of vectors. Unfortunately, it is not easy to feed vectors continuously into Kung's linear array without complicating the cell's function. Furthermore, the scalar throughput rate is fixed with a given cell speed. Therefore, it takes more time to generate a long vector than a short vector. Hence, direct application of this linear array cannot meet the real time requirement and some modification has to be made.

One way to modify this architecture is to have several linear arrays working on successive input blocks in parallel. However, the data flow pattern will be very complicated; hence, this will destroy the simplicity of the systolic array, which is the vital part of the systolic approach. Another method of modification is to exploit Kung's hexagonal structure for matrix-matrix multiplication. Several input vectors can be fed into this array at the same time as a single matrix. However, this structure is designed for performing the matrix multiplication only once, which is very common in most structures in computer science area, and hence the continuous usage of this structure for a large number of input vectors is not considered.

### 3.4.1. Implementation of Block State Filters

#### 3.4.1.1. Matrix-Vector Multiplication

A very efficient structure for realizing the block state filter can be constructed by interconnecting cells which have an internal structure as shown in Figures 3-3. Kung's hexagonal two dimensional array looks unnatural for a digi-

tal filter implementation, since both coefficients and input data travel through this array. In digital filter design, however, the same set of coefficients is going to multiply a large number of input samples; hence, it is more reasonable to keep the coefficients fixed in cells, thereby eliminating an unnecessary (and expensive) communication.

Before showing how to realize a block state filter, we will demonstrate how these cells can be applied to multiplying a  $3 \times 4$  matrix by vectors. This multiplication at time  $n$  can be expressed as in equation (3-7)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \begin{bmatrix} X_{n,1} \\ X_{n,2} \\ X_{n,3} \\ X_{n,4} \end{bmatrix} = \begin{bmatrix} Y_{n,1} \\ Y_{n,2} \\ Y_{n,3} \end{bmatrix} \quad (3-7)$$

where  $X_{n,i}$  is the  $i^{\text{th}}$  component of vector  $X_n$ . Figure 3-8 shows the structure of an array to implement this operation. The coefficient stored in each cell is the value in the corresponding position of the matrix.

At time  $n$ , the input vector  $X_n$  enters this array from the top with each component aligned with its corresponding column. Each element in this input vector is actually skewed in time, and this time skew also happens to the output samples. A commutator model at this input is appropriate to represent the input data flow. The speed of this commutator is synchronized to the input sampling rate, and the same commutator can be used at the output to convert the output from vectors to scalars. Latches are inserted on some input paths to compensate the time skew among the cells on the first row in the array. Except the cells on the bottom row, each cell transmits its upper input to the cell below directly without any operation. It also updates its left input according to the function shown in Fig. 3-3a, and then sends the updated value to its right cell. Hence, each cell, in addition to transmitting input data to its lower neighbor, performs one step of updating of the inner product between the input vector

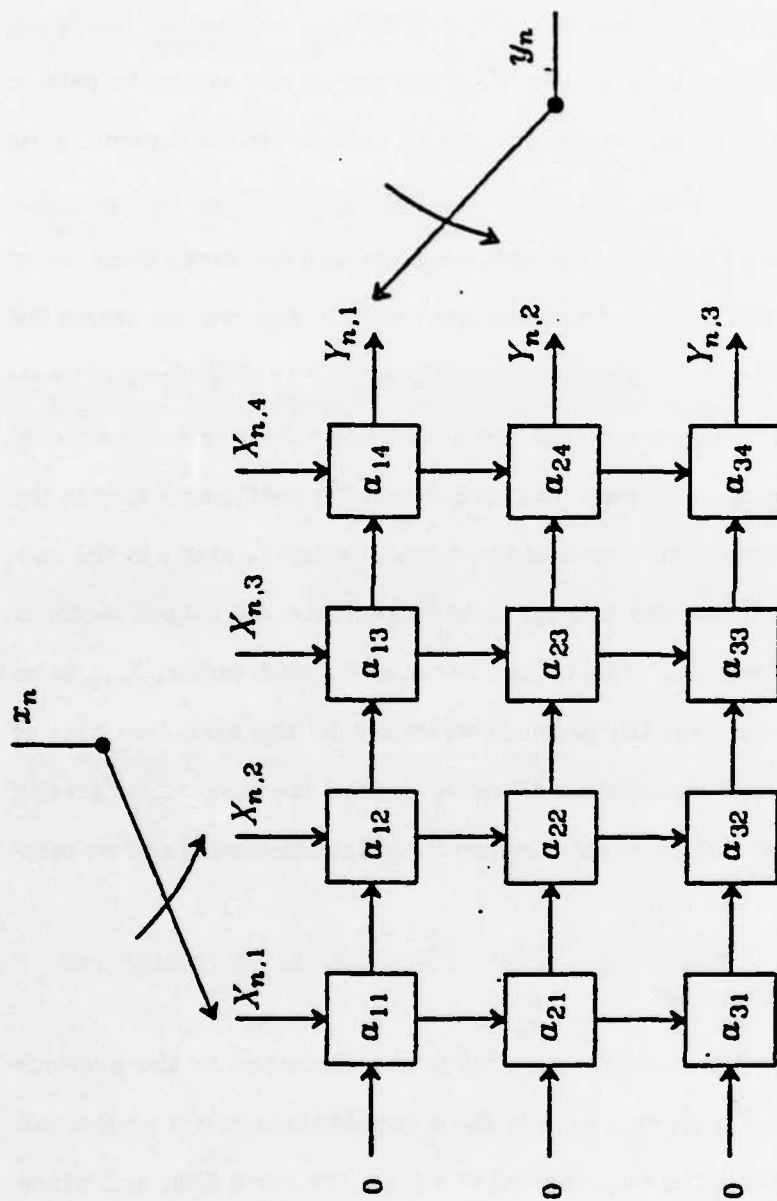


Figure 3-8 Two Dimensional Systolic Array for Matrix-Vector Multiplication



and its corresponding row vector in the matrix. After the cells on the rightmost column finish their updating, their right outputs constitute the final output vector  $Y_n$ .

The basic principle behind the use of this structure is that when the upper left cell finishes its computation on the  $n^{\text{th}}$  input vector and sends the data to its two neighbors, this cell can work on the  $(n+1)^{\text{th}}$  vector. In the meantime, its two neighboring cells can start computing the  $n^{\text{th}}$  vector. Thus, the computation time of  $X_n$  forms a "wavefront" propagating through the array from upper left to lower right. When the wavefront reaches the right end, we can obtain the final output vector. This phenomenon looks very similar to S. Y. Kung's "Wavefront Array Processor"[24], except that the data transmission is unidirectional and only the data propagate through the array while the coefficients stay in the cells. Although the three components of the output vector  $Y_n$  arrive at the output ports at different times, the average time to generate one output vector is the execution time of one cell. This is true because the next vector,  $Y_{n+1}$ , is on the next wavefront which lags the previous wavefront by the execution time of one cell. For the vertical transmission of the input data, the time can be greatly reduced, since the operation is simply transmitting data downward and no computation is required.

#### 3.4.1.2. Filter Implementation

For the matrix-vector multiplication structure described in the previous subsection, the vector throughput rate is fixed and depends solely on the cell speed. This feature meets the requirement of a real time block filter and hence is very helpful in realizing the filter.

Figure 3-9 shows the block diagram of a block state filter. The small squares labeled A represent the computation of the feedback state matrix. The three big rectangles B, C and D perform the matrix-vector multiplications. The



structure of these three rectangles is similar to that in Figure 3-8, but with different sizes. Matrices B and D are initialized to zero at their y inputs, and matrix C is initialized by the output from matrix D.

In matrix B, the input vectors are transmitted vertically and their output samples are propagated horizontally to its right. These output samples are used by matrix A to calculate the next state vector  $R_{n+1}$ , which in turn, will be used by matrix C to calculate the final outputs. As for matrices C and D, the input samples  $R_n$  and  $X_n$  are transmitted horizontally, and the output are propagated vertically to the bottom.

If combining four cells in Figure 3-8 into one PE, the new structure can have the same form as before and the execution time of each PE will be exactly the same as that of the PE's operating on the feedback matrix. Hence, if using the same PE for both the feedback and feedforward matrices, the real time condition will be automatically satisfied. Because the cells are equally loaded computationally, the input data can be kept pumping through the whole structure and no idle time is necessary anywhere in this structure.

If working synchronously, the output samples between two adjacent rows will be separated by the execution time of one cell. Since all the cells that work on matrix A operate independently, this time skew on the outputs from matrix B won't affect their function. Thus, the output samples from matrix A are also skewed by the same time interval. This time skew matches perfectly to the operations in matrix C, since the starting time of the cells on the same column is also skewed by this time. This perfect match also applies to the interface between the output from D and input of C.

If a vertical bus is used for transmitting the input sample on each column of matrix B, all the output samples in the same vector can be generated at the same time. However, the time skew of matrix C between cells on each column is

still necessary. Therefore, different number of buffers is required to store the output samples from A to reflect this time skew. On the other hand, if a bus is used on each row of both matrix C and matrix D, the propagation wavefront will be horizontal and moves vertically. Hence, the interface between C and D is still perfectly matched and output samples in the same vector can be generated at the time.

As mentioned before, the throughput rate can be increased by simply increasing the block size. This increased block size results in larger rectangles and more cells. However, the speed of each cell does not have to be faster. For matrix A, neither the number nor the speed of cells has to change to cope with the higher speed. Thus, we can effectively trade hardware with speed without difficulty. And real time processing is possible even with low speed processors, if using more hardware is not a problem.

### 3.5. Some Applications

Besides recursive filtering, the efficient structure of block state filters can also be applied to some other areas. In this section, the applications to computer graphics and decimation and interpolation are presented. The rotation, scaling and translation of computer graphs can be represented by a  $4 \times 4$  matrix multiplying the input vectors. An input vector is composed of the three coordinates of a point in the graph plus an extra parameter. Hence, the two dimensional systolic array for implementing a matrix-vector multiplication can be applied to the computer graphics. Multistage FIR filters are considered as efficient structures to realize the decimation as well as interpolation. This is true due to the fact that FIR filters are feedback free, and hence decimating the output samples is easy. IIR filters, on the other hand, are considered inferior to FIR filters due to their feedback operations. However, in the block state filters, the state variables rather than the output samples are fed back for the

calculation in the next cycle. Hence, the direct sampling rate reduction on the output samples is possible. On the other hand, the computation of all the state variables is necessary.

### 3.5.1. Computer Graphics

Interactive computer graphics becomes more and more important in fields like computer aided design, man/machine communication and character recognition etc. In some sense, computer graphics is very similar to digital signal processing systems. Both have a large number of input samples to be processed and both require high throughput rate. The two-dimensional systolic array architecture described earlier can also be applied to computer graphics in some applications, such as graph rotation, scaling and translation.

A three-dimensional (3-D) graph is a collection of many points with three coordinates. Hence, each point can be represented by three numbers  $(x,y,z)$ . Rotation and scaling of a graph can be represented by a  $3 \times 3$  matrix multiplying all the points in the graph[25]. Scaling a graph by a constant  $S$  is simply multiplying every component by this constant. The coordinates after the scaling  $(x',y',z')$  can be related to the coordinates before scaling  $(x,y,z)$  by

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & S \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The rotation of a graph in three dimension can be decomposed into a combination of three rotations about  $x$ -,  $y$ - and  $z$ -axis respectively. Rotating an angle  $\vartheta$  about the  $z$ -axis can be represented as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta & 0 \\ \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The matrices representing the rotation about  $x$ -axis and the  $y$ -axis can be written as

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}$$

The complete rotation can be represented by a single 3x3 matrix which is the product of the above three matrices with proper angles. Hence the combination of scaling and rotation can be modeled by a full 3x3 matrix in general.

Translation of an object in the three dimensional space by  $D=(D_x, D_y, D_z)$  can be done by adding the three components of  $D$  to the corresponding component of each point of this object. Hence, the position  $(x', y', z')$  after translation can be represented as

$$x' = x + D_x \quad y' = y + D_y \quad z' = z + D_z$$

Unfortunately, this operation cannot be represented by a 3x3 matrix multiplying a vector as in the rotation and scaling cases. However, introducing after  $z$  an extra coordinate, which is always 1, the translation can be represented as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & D_x \\ 0 & 1 & 0 & D_y \\ 0 & 0 & 1 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Adding one more column and row to the matrix, the rotation and scaling can also be represented in the same dimensionality as in the translation case. A general matrix representation of a combination of any number rotations and scalings can be written as

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If also including any number of translations, the overall operation can be represented as the following matrix

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A highly pipelining structure can be obtained by using a  $4 \times 4$  systolic array as shown in Figure 3-8. Furthermore, since the last coordinate is always 1 for each point, the same function can be realized with a  $3 \times 4$  systolic array. Also because the last entry of each row is always multiplied by unity, the structure can be reduced further to an array of size  $3 \times 3$  with the three entries in the last column as the initialization values entering this array from the left. The structure is shown in Figure 3-10. The throughput rate of this structure again is governed by the time for one cell to perform one addition and one multiplication.

The difference of this application from the IIR filter is that the entries will have different values for different transformation. Therefore, an efficient way to adapt these coefficients is essential. Furthermore, multiplying matrices together for a combination of rotation, scaling and translation, requires matrix-matrix multiplications. Hence, another efficient structure for implementing this operation is also essential. Thus, some further research in this area should be done before the systolic array can be applied to the transformation of graphs.

### 3.5.2. Decimation and Interpolation

Decimation and interpolation are linear and periodic time varying systems. The period depends on the decimation or the interpolation ratio. This is true because almost all signal processing systems are modeled as scalar input scalar output systems. Systems with different sampling rates at the input and output cannot be represented as Linear Shift Invariant (LSI) SISO system functions. However, they can be represented as LSI MIMO systems with a proper ratio between the input and the output block sizes[26, 27].

Sampling rate reduction, or decimation, can be realized in two stages. First, pass the input sequence through a low pass filter with bandwidth  $\frac{\pi}{M}$ .



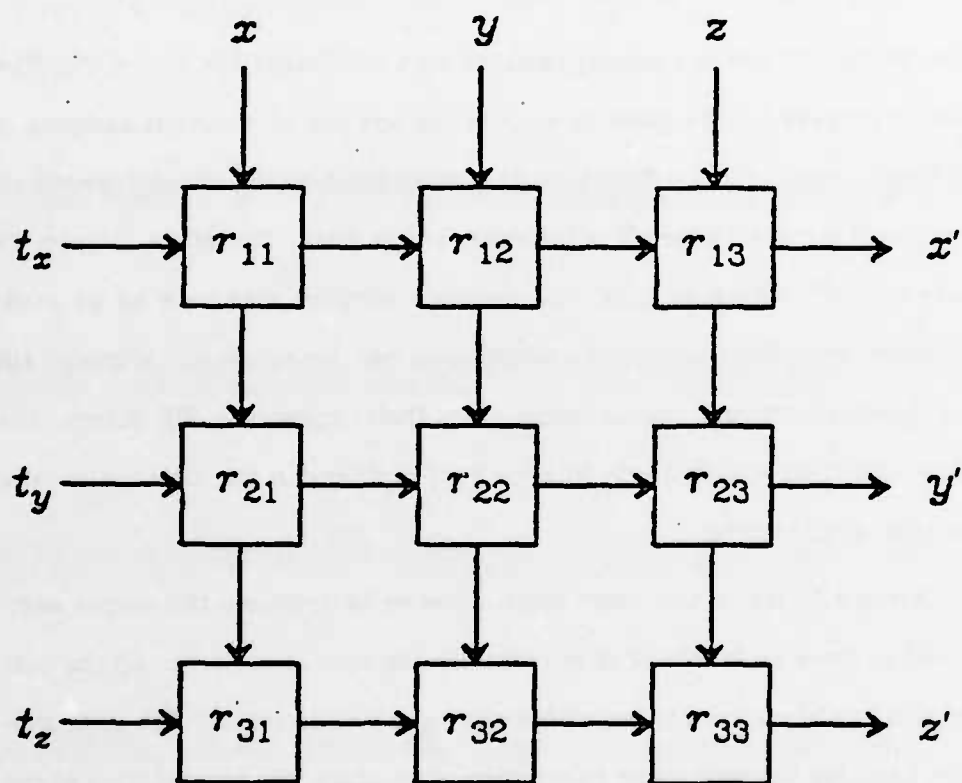


Figure 3-10 Two Dimensional Systolic Array for Computer Graphics

Then, throw away  $M-1$  out of  $M$  samples, retaining the  $M^{\text{th}}$ , where  $M$  is the decimation ratio. The low pass filter is used to prevent aliasing resulted from the lower sampling rate. Interpolation, on the other hand, can be realized by inserting  $I-1$  zeros between each adjacent samples and then passing this new sequence through a lowpass filter with bandwidth  $\frac{\pi}{I}$ .  $I$  is the interpolation ratio and the lowpass filter is used to remove the spurious components generated from the interpolation process. The detailed design procedure can be found in[29].

The low pass filters are usually realized with multistage FIR filters[29]. The decimation process can be done by calculating one out of  $M$  output samples, if an FIR filter is used. For IIR filters, on the other hand, each output depends on the previous  $N$  samples, where  $N$  is the order of the filter. Therefore, in order to calculate the  $M^{\text{th}}$  output sample, the previous samples also have to be computed; hence, very little saving in computation can be achieved. Although IIR filters require much less computation than their equivalent FIR filters, the recursive operation makes them inferior to FIR filters in the decimation and interpolation applications.

Block state filters on the other hand, allow us to decimate the output samples directly, since no feedback is occurred to the output samples. All the output samples in a block are generated from the same state vector. The state vector is fed back for the next cycle calculation. Therefore, the computation of the state variables cannot be reduced. Even if the block state filters have a little higher computational rate, the simplicity in the implementation of their structures makes them still very attractive.

Suppose the low pass filter with bandwidth  $\frac{\pi}{M}$  is realized with a block state form with block size  $M$  whose representation is rewritten in equation (3-8).

$$R_{n+1} = AR_n + BX_n$$

$$Y_n = CR_n + DX_n \quad (3-8)$$

The sizes of the four matrices A, B, C and D are respectively,  $N \times N$ ,  $N \times M$ ,  $M \times N$  and  $M \times M$ , where  $N$  is the order of the filter. Since the output vector  $Y_n$  is not fed back, the output decimation can be done by throwing away  $M-1$  samples in each output vector, which is equivalent to calculating only the first sample in  $Y_n$ . Therefore, only the first rows of matrices C and D are required to compute the output samples. Since matrix D is lower triangular, only the leftmost entry of the first row is non-zero. Hence, this matrix multiplication is reduced to a scalar multiplying the first samples of the input blocks.

With the sizes of all these matrices in mind, we can calculate the number of multiplications per input sample, which is a good indicator of the speed of this implementation. For an  $N^{\text{th}}$  order filter with all complex poles, this number is

$$\frac{(2N + NM + N + 1)}{M} = N + \frac{3N + 1}{M} \quad (3-9)$$

This number is compared to that of an optimal multistage FIR filter approach developed by Crochiere *et al.* [30], and the result is shown in Table 3-1.

	One Stage (Block State)	Two Stage (Block State)	Two Stage (FIR)	Three Stage (FIR)
Decimation Ratio	$D_1=100$	$D_1=20$ $D_2=5$	$D_1=50$ $D_2=2$	$D_1=10$ $D_2=5$ $D_3=2$
Filter Order	$N_1=14$	$N_1=4$ $N_2=14$	$N_1=423$ $N_2=347$	$N_1=39$ $N_2=39$ $N_3=356$
Mult/Sample	14.43	5.48	5.98	4.06
Data Storage	114	43	769	436

Table 3-1 Comparison between block state and FIR filters for Decimation

Zeman [31] mentioned that block state filters can save computation only if the decimation ratio  $M$  is moderate, however, from Table 3-1, for a decimation ratio of 100, the two stage block state filter is comparable to the two stage optimal

FIR filters. As for the implementation consideration, block state filters are much simpler than FIR filters and their inherent parallelism can result in very high speed. Notice also the huge difference of the data storage requirement between both approaches. The multistage FIR filter approach requires a memory space whose size is an order of magnitude higher than that of a block state filter.

Another advantage of the block state filter as well as the optimal FIR filter over many other structures is that the decimation ratio can be any integer. It does not have to be a number of a power of 2. Half band digital filter[32] and wave digital filter approaches, for instance, do require this constraint.

The block state filter can also be effectively applied to the interpolation. Suppose the filter is realized with a block size  $l$ , where  $l$  is the interpolation ratio. Since the input sequence has  $l-1$  zeros out of  $l$  samples, some savings in computation can be made. Matrices  $B$  and  $D$  are the only two terms operating on the input vectors. If choosing the first samples to be non-zero, only the first columns of  $B$  and  $D$  are required. This reduces the number of multiplications of these two matrices from  $Nl + (l+1)l/2$  to  $Nl$ . Furthermore, this nonzero entry in the input vectors can be any component in the vector. If choosing the last element, the operation of matrix  $D$  again reduces to a scalar multiplication. The average computation then equals that of equation (3-9) with  $M$  replaced by  $l$ .

Both decimation and interpolation can be represented by a single block filter representation with a proper ratio between the input and output block sizes. In the scalar filter modeling, decimating the output samples and inserting zeros in the input sequence cannot be modeled as a linear time invariant process. Hence, these processes have to be modeled separately from the filtering operation. However, as shown above, it is straightforward to model both decimation and interpolation by a single LSI MIMO filter.

### 3.6. Conclusions

A filter structure, which can achieve any desired speed with localized interconnection has been presented in this chapter. This structure, modeled by state equations in a block form, utilizes extensive pipelining and concurrency. State equations require fixed computational rate for the recursive operation, even when the block size changes, since the state variables rather than the output samples are the feedback elements. The number of states equals that of the poles of a filter, and hence is a fixed number for a given filter. Any desired speed can be achieved by choosing a proper block size so that the state feedback operation can be completed in time. Buffering the input samples also increases the parallelism which makes pipelining and concurrency feasible. Systolic arrays for the matrix-vector multiplications provide a very simple interconnection which makes this structure very attractive in the VLSI application. Furthermore, this structure can actually be applied to any linear shift invariant systems, which can be represented by the discrete time state equations.

The applications of this structure to computer graphics as well as decimation and interpolation are also presented. The regularity of the interconnection in systolic arrays and the lower computational rate in IIR filters compared to the equivalent FIR filters make this structure very attractive in these two applications. However, the coefficient adaptation in computer graphics must be solved before this structure can be effectively applied. This is also true for most time varying systems which are modeled by time varying state equations.

## CHAPTER 4

### PERFORMANCE ANALYSIS

This chapter compares the performance of the realization discussed in chapters 2 and 3 in various respects. The performance of the best known cascade and parallel connection of second order filters will be discussed first in every respect as a reference. The performance of the other structures will be treated afterwards.

Since real time is of prime concern in most filters design, the limitation of realizable sampling rate is discussed in the first section with the assumption of no message transmission delay. The effect of communication delay on this rate is then treated, as well as the latency of the signal through the filter. A detailed comparison of the computational rate of FIR and IIR filters will then follow. This will indicate that the IIR filter is best for processing signals with low to moderate high sampling rate, and FIR filters are best for extremely high sampling rate. Finally, the percentage of idle time for the PE's will be discussed to investigate the efficiency of the hardware usage.

A detailed discussion of the performance of a block state filter with a block diagonal state matrix was given in the sections 3.2, 3.3 and 3.4. An even more complete analysis will be given in this chapter and its performance will be compared to other structures mentioned in chapter 2. After finishing the analysis and comparison in this chapter, it will become clear why this structure outperforms any other structure in almost all respects.

#### 4.1. Speed Limitations

In this section, the highest achievable throughput rate for every structure

will be discussed. We will initially assume no delay for message transmission between PE's. The effect of transmission delay will be treated in the next section.

Although requiring more computations than SISO filters, block processing of the input samples can in theory achieve any desired sampling rate. Real time is guaranteed by choosing an appropriate block size. This size depends on the input sampling rate and the speed of the PE's. In practice, however, the sampling rate is limited by the filter latency and the buffer size. The highest sampling rate for SISO filters, on the other hand, is limited by the PE speed. If the sampling rate is higher than this limit, faster PE's are necessary. Hence, for signals sampled at very high rate, block processing is a better approach.

#### 4.1.1. Parallel and Cascade Forms

The throughput of both the parallel and cascade forms is, in general, bounded by the throughput of one second order filter, no matter how complex the original filter is. For the cascade structure as in Figure 2-1 section 2.2.1, the pipelining processing of the input samples gives us a high throughput rate, which depends on the longest processing time among all the cascaded stages. If each stage is either a second or a first order filter, the throughput rate depends on that of a second order filter.

For the parallel structures as shown in Figure 2-2 section 2.2.2, since all the second order filters work on the input samples simultaneously, the throughput rate also depends on the longest processing time among all sections. If the final summing node is lumped into one of the filter sections, the longest processing time is the processing time of a second order filter plus that of the final summing node. If this node is cascaded with all the filter sections, the throughput rate then depends on a second order section if the summing node can be processed faster than a second order filter. The difference between these two



structures is that the cascade form has longer latency but simple communication links, while the parallel form has several-to-one type of communication at the final summing node. The time bound for both forms is the time to perform five multiplications and four additions, if a direct form implementation is used for each section.

#### 4.1.2. Systolic Arrays

Kung's systolic array[7] also uses extensive pipelining but in a different manner from the cascade form. In this array, input and output samples are traveling in opposite directions. The output samples are also fed back to this array, and travel in the same direction as the input samples. Thus, this structure can be viewed as three sequences pipelined through each cell at the same speed. Due to this pipelining nature, the throughput rate depends on the execution time of one cell, which contains two multiplications and two additions. Since, as noted in Figure 2-4 in section 2.3, every second cell is idle at any given time, the actual processing time per output sample is two times that of one cell. Thus, the minimum sampling period achievable is twice that of the execution time of one cell, which contains only four multiplications and four additions. Therefore, this structure can achieve a slightly higher throughput than either the cascade or the parallel form.

The advantage of this structure over the cascade and the parallel forms is that it can be realized directly from the filter difference equation. No factorization or partial fractional expansion is required. This simplifies the design procedure. Furthermore, similar to the cascade form, this structure requires only local interconnection. It does not require any broadcasting type of communication as in the parallel form. However, the roundoff noise behavior is similar to that of the direct form structure, which is a serious problem compared to the other two structures.

#### 4.1.3. SSIMD mode

Although Barnwell's structure can achieve higher sampling rate than that of the three forms mentioned above, the highest rate of this structure is still limited. The average time to generate one output sample, in this mode, equals the time to do one multiplication and addition[9]. This time bound is about one fifth of that of the parallel and cascade structures, if using the same PE's. Thus, Barnwell's algorithms can achieve a sampling rate five times faster than those two structures.

To write the programs in SSIMD mode, the only information required is the length of the program, the time at which recursive inputs are needed and recursive outputs are available. Hence, a task with a single recursive output as in Figure 2-5b in section 2.5 can be characterized as

$$K(I(L), I(L-1), \dots, I(1); R; T)$$

where  $K$  is the task identifier,  $T$  is the task length,  $R$  is the output time for the recursive output,  $I(l)$  is the input time for the  $l^{th}$  delayed recursive output, and  $L$  is the value of the longest delay. Some important theoretical results for this realization are summarized as follows[9].

- (1) All PE's are started at equal intervals and the outputs are periodic with period  $t_m = \frac{mT}{M}$ , where  $M$  is the number of PE's used. This is true only if  $M$  is less than some number  $M_x$ .
- (2) The upper bound in (1) is given by

$$M_x = \text{INT}[M(L_x)] = \text{INT}\left[\min_i [M(l)]\right] \quad (4-1a)$$

$$M_x = \text{INT}\left[\min_i \left\lceil \frac{lT}{I(l)-R} \right\rceil\right] \quad (4-1b)$$

where  $M(l)$  is the non-integer number of PE's which could be utilized if the only constraint came from the recursive input  $I(l)$ ,  $L_x$  is the value of  $l$  for which  $M(l)$  is minimum, and  $\text{INT}[\cdot]$  means "the integer part".

- (3) The greatest throughput achievable is obtained with a time skew of  $t_z = \frac{R - 1(l_z)}{l_z}$ . This solution is generally achieved with  $M_z + 1$  PE's if  $M(l_z)$  is not an integer. Adding one more PE will introduce some idle time in each PE; in other words,  $T$  will increase.

Based on these results, it is easily seen that given a signal flow graph, the maximum number of PE's is immediately available, and the maximum throughput rate is the function of PE speed only.

According to the first property, the more PE's that can be used without increasing  $T$ , the higher the throughput rate this structure can achieve. However, it is not possible to increase the number of PE's indefinitely. According to property 2, the sampling rate saturates when this number reaches some value. Beyond this value, increasing the number of PE's does not help improve the speed performance. This critical number, which equals to  $M_z + 1$ , can be obtained by examining the throughput while increasing the number of PE's with the assumption that all the input samples are stored somewhere and are ready for use. This assumption ensures us that the speed limitation actually comes from the structure itself but not from the finite input sampling rate. The number, at which the sampling rate starts saturating, is the maximum number of PE's can be used to increase the throughput rate.

The previously summarized results in this section will be verified by considering the limitations on a specific PE. Since it is single instruction multiple data mode, all the other PE's will undergo exactly the same situation. This specific PE is examined in the following two steps. First, the starting time constraint of this PE imposed upon by the availability of the  $l^{\text{th}}$  delayed recursive input will be considered. This number is obtained by delaying the starting time until no waiting time is necessary for this PE. Then, this limitation will be generalized by considering all the delayed recursive inputs. If each delayed recursive input

gives rise to different constraint of starting time, the latest time is chosen in obvious fashion.

Suppose PE 0 starts working at time 0 and receives all the inputs in time. This PE can, then, transmit its recursive output at time R and the final output y at time T. This recursive output will be used as the  $l^{\text{th}}$  delayed recursive input by some other PE  $l$ . The earliest time that this  $l^{\text{th}}$  PE can start working is at time  $R-l(l)$ . This is clear from the graph shown in Figure 4-1. If it starts earlier, it cannot receive the output from PE 0 in time; hence, some idle time occurs. The reason that this receiving PE is labeled as PE  $l$  is that there must be another  $l-1$  PE's which use this recursive output as their  $1^{\text{st}} \rightarrow (l-1)^{\text{th}}$  delayed recursive input. These  $l-1$  PE's can start working before this receiving PE, obviously. Hence, the time skew between any two adjacent PE's is  $\frac{R-l(l)}{l}$ .

In the next cycle, PE 0 will start working following the last PE. The time interval between the starting time of these two PE's is also  $\frac{R-l(l)}{l}$ . Since PE 0 will resume its function at time T, if it can start working on or before T in the next cycle, and no waiting time will occur. The maximum number of PE's  $M(l)$  that can be used without introducing idle time must satisfy the following relation

$$M(l) \cdot \frac{R-l(l)}{l} \leq T$$

or

$$M(l) \leq l \frac{T}{R-l(l)}$$

Since it is unlikely that  $\left\lfloor \frac{lT}{R-l(l)} \right\rfloor$  is an integer,  $M(l)$  can be expressed as

$\text{INT} \left\lfloor \frac{lT}{R-l(l)} \right\rfloor$ . If considering all  $l$  from 1 to L, equations (4-1a, b) are immediately available. This verifies properties (1) and (2).

If  $M_s$  PE's are used to realize the filter, no PE has to sit idle. This realization is called PE optimum, since a high rate filter can be realized using a max-

Processor No.

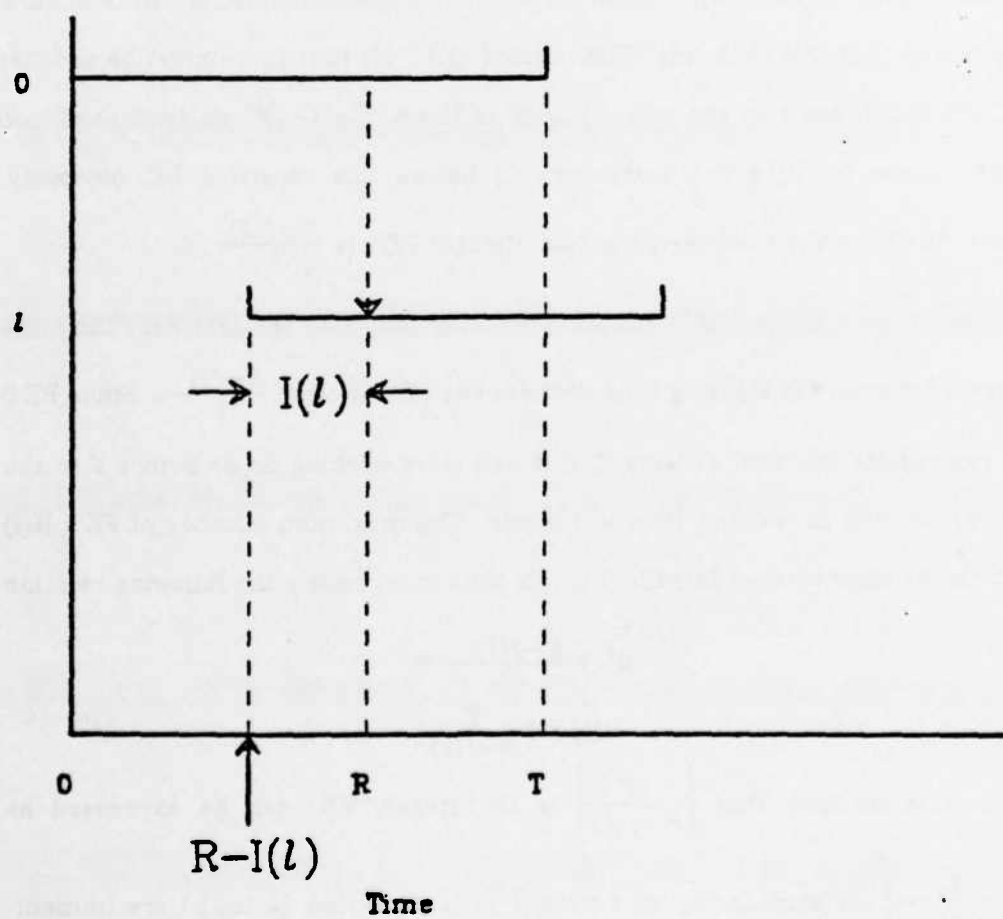


Figure 4-1 Timing Diagram of SSIMD Structure

imum possible number of PE's without introducing any idle time. However, if  $\frac{lT}{R-I(l)}$  is not an integer, introducing an extra PE can achieve a slightly higher sampling rate. This realization is called speed optimum, since no other realization in the SSIMD mode can achieve a higher sampling rate. On the other hand, since idle time exists, it is not PE optimum.

From Figure 4-1, the shortest possible time skew between any two adjacent PE's is  $[R-I(l_z)]/L_z$ . If using  $M_z+1$  PE's, PE 0 will sit idle for a while before it can start the computation for the next cycle. Since this algorithm is SSIMD, this idle time occurs to every PE and hence, lowers the efficiency of the usage of the PE's. However, each PE can start working right after it receives data from some other PE's, whereas with  $M_z$  PE's, each PE has to wait until the previous cycle finishes even if the input data is ready for it to use. With another extra PE, each PE has to wait a whole cycle and hence the throughput rate does not increase with this PE. Hence, one extra PE lowers the PE usage efficiency, but achieves the highest possible sampling rate. This verifies property (3).

Figure 4-2 shows the computation sequence of one PE to realize a third order filter. It is obvious that the execution time between any two delayed recursive inputs is the time to perform one multiplication and one addition. However, in the actual implementation, the time to transfer the data between the PE's and the I/O ports should also be included in this time skew. Therefore, the throughput rate is actually governed by the time for fetching one sample from the input port, sending one sample to the output port, and the time to perform one multiplication and one addition. This data transfer time should also be added to the other two structures mentioned in sections 4.1.1 and 4.1.2.

#### 4.1.4. Block I/O Filters

Burrus mentioned in his paper[33] that the block filter is a better choice

$$\begin{aligned}
 r_n &\leftarrow -z_n \\
 r_n &\leftarrow r_n + b_1 \cdot r_{n-1} \\
 r_n &\leftarrow r_n + b_2 \cdot r_{n-2} \\
 r_n &\leftarrow r_n + b_3 \cdot r_{n-3} \\
 y_n &\leftarrow a_0 \cdot r_n \\
 y_n &\leftarrow y_n + a_1 \cdot r_{n-1} \\
 y_n &\leftarrow y_n + a_2 \cdot r_{n-2} \\
 y_n &\leftarrow y_n + a_3 \cdot r_{n-3}
 \end{aligned}$$

Figure 4-2 Computational Sequence of one PE for the Barnwell Algorithm

than the direct form implementation only if the order of the filter exceeds 25. This number was obtained by comparing the number of computations in both structures. Since the number of operations per output sample is a good indicator for the speed performance of only serial processing systems, his conclusion does not apply to the case of parallel processing. For parallel processing, the sampling rate depends not only on this number but also on the parallelism in the algorithm. The sampling rate can even be totally irrelevant to this number, as in the block state filter implementation described in section 3.4. For multiprocessing, it is advantageous to realize in block form, even for low order filters, if using a large number of PE's is of minor concern.

For block filters, Burrus assumed that the computation is divided into two matrix-vector multiplications associated with  $B_1 Y_n$  and  $A_1 X_n$  and two convolutions associated with the operations of  $B_0^{-1}$  and  $B_0^{-1} A_0$ . These two convolutions are computed by FFT algorithms. Although the FFT can reduce the computational rate, block processing usually increases this rate. Furthermore, although the data flow of the FFT is pipelining in nature, the highly complex data flow pattern between adjacent pipelined stages makes local connection very difficult; hence, we will implement the product matrices directly instead of dividing into



two stages. Therefore, the implementation of a block I/O filter will be treated as three matrix-vector multiplications.

As noted in section 3.2, the maximum possible throughput rate depends on how the feedback product matrix  $B_0^{-1}B_1$  is realized. The implementation of this feedback matrix and the criterion for real time processing will be discussed in this section. Subsequently, the feedforward matrices will be treated. Obviously, the two dimensional systolic array discussed in section 3.4 can be employed for the implementation of these feedforward matrices. However, the function of each cell depends on the implementation of the feedback matrix. Since this array uses extensive pipelining, it cannot be applied to the feedback operation.

Since the number of non-zero columns of the feedback matrix is invariant as the block size  $L$  changes, the number of operations in each row stays unchanged. If each row is assigned to one PE, the computational rate in each PE does not change with the block size. For an  $N^{\text{th}}$  order filter, it takes  $N(t_s + t_m)$  to compute this feedback operation, where  $t_s$  and  $t_m$  are defined as in section 3.2. Real time condition is satisfied, if the following equation holds

$$L \geq \frac{N(t_s + t_m)}{T_s} \quad (4-2)$$

For a given filter, a finite  $L$ , which satisfies this equation no matter how large  $t_m$  and  $t_s$  are and how small  $T_s$  is, always exists. In other words, a real time filter is always realizable for signals sampled at very high rate on even very slow hardware. Note also that only the lower right  $N \times N$  submatrix is the feedback term, because all the outputs from the other rows are multiplied by zeros in the next block period.

In the operation assignment mentioned above, more links are needed to connect the PE's when the block size increases. Certainly many other methods can be found for the implementation. It is possible to either divide each row into more PE's or to group more rows into one PE. The former method

complicates the communication environment and the latter requires larger block size and hence longer latency. Grouping all the lower  $N$  rows into one PE would be a good way, as far as communication complexity is concerned, if the much longer latency is tolerable. This increased latency might severely restrict the applicability of this algorithm.

As for the implementation of the other two feedforward product matrices, the only constraint is that the execution time of each PE should not be larger than  $LT_s$ , where  $L$  is decided by the implementation of the feedback matrix. The structures described in section 3.4 to implement the feedforward matrices can also be used to realize this feedforward operation. However, each PE in this structure should contain the same amount of operation as that of the feedback PE's so as to fully utilize the hardware. This is not an easy task, because of the dissimilarity among the structures of the three matrices. Furthermore, reducing the communication paths in the feedforward matrices only makes little sense, since the throughput suffers more from the complex interconnection in the feedback operation as compared to the feedforward operations.

#### 4.1.5. Block State Structures

Within the context of block state filters, we have presented both two dimensional systolic arrays and block cascade and parallel approaches. The block cascade and parallel filter, a simple extension of the scalar cascade and parallel forms, is going to be studied in this section and compared to the systolic approach described in section 3.4.

##### 4.1.5.1. Systolic Arrays

As described in section 3.2, the block state filter can achieve any desired sampling rate by simply selecting an appropriate block length. This is true for either a full or a block diagonal state matrix. The advantage of the block diago-

nal form over the full matrix is that it requires much shorter block size to achieve the same sampling rate. This implies that it can usually realize the same filter with fewer PE's while suffering less from the filter latency. Another advantage of the block diagonal state matrix is that the filter can be easily realized with a systolic approach, which requires localized interconnection only.

#### 4.1.5.2. Block Cascade and Parallel Forms

Unlike the other block filters described in sections 2.5, 2.6 and chapter 3 that have the property that any throughput rate can be achieved by simply increasing the block size, Zeman and Lindgren's structures have an upper bound on the achievable sampling rate. In other words, real time cannot always be satisfied by increasing the block size. The reason is that the internal operation of a second order filter is serial rather than parallel. The increased parallelism generated from buffering the input samples is not fully utilized in these structures.

Similar to the scalar cascade and parallel forms, the overall throughput rate depends on the throughput of a second order filter section. Also due to their serial processing within each second order section, the overall throughput depends on the average speed of an inner product arithmetic unit to generate one output sample. Apparently, this speed is inversely proportional to the average number of multiplications required to generate one output sample. This number, in turn, depends on the block size  $L$  and the order of the filter  $N$ . Considering that the  $D$  matrix is lower triangular, the total number of multiplications for generating one output sample is :

$$\frac{1}{L} \left[ N(2+L) + L \left[ N + \frac{L+1}{2} \right] \right]$$

Differentiating this number with respect to  $L$  and setting it to zero, the optimal block length, which gives us minimum number of operations, can be obtained.

The optimal block length is

$$L_{opt} = 2\sqrt{N}$$

The number of operations associated with this block length is

$$2N + 2\sqrt{N} + 0.5 \quad (4-3)$$

Hence, the maximum achievable sampling rate with this architecture is bounded by the number in (4-3) and the chip speed.

This bound on the sampling rate can be removed if each second order filter is realized in parallel with a systolic approach. However, compared to the systolic implementation in section 3.4, the block cascade form has longer latency because of its pipelining nature. Both structures require more computation than the systolic structure described in section 3.4 due to the fact that D matrix exists in each second order subfilter, and each one has the same size as the D matrix in section 3.4. The total computational amount, on the other hand, is the same for the other three matrices in each structure.

#### 4.2. Susceptibility to Transmission Delay

In this subsection, the effect of the message transmission delay on the overall sampling rate is investigated. It is well known[34] that the feedforward operation can be put into pipelining stages and any number of delays can be inserted without affecting the overall sampling rate. Hence, the delay on the feedforward path will be neglected. However, any communication delay does increase the filter latency. The effect of delay on any recursive computation will be carefully investigated in this section. If any delay exists in the data transfer among the PE's computing the feedback operation, the speed performance will be degraded.

##### 4.2.1. Cascade and Parallel Forms

The cascade form is a typical pipelining structure; hence any transmission

delay on the links connecting the adjacent stages does not affect the overall sampling rate at all. For the parallel form, two sets of communication links are required for the data transmission. One set distributes the input samples to each second order section. The second set of links transmits the results of each section to the final summing node. If this node is cascaded with all the other filter sections, the transmission delay does not have any effect. On the other hand, the delay slows down the filter if the summing node is lumped into one of the subfilters. Furthermore, the filter section that contains the summing node has the longest execution time, and hence dominates the throughput. If the connection of the summing node is cascaded, transmission delay on any link does not affect the throughput rate. Since the whole structure can be viewed as a connection of three blocks, namely, data source, filter sections and the final summing node, the transmission time between any two cascading stages does not affect the sampling rate.

#### 4.2.2. Systolic Arrays

For the systolic arrays, any transmission delay would reduce the speed performance. Although the systolic array is a highly pipelined structure, the data are pipelined in two directions for the IIR filter whose structure is shown in Figure 2-4 in section 2.3. The behavior of this two way pipelining is quite different from the pipelining in the cascade form. Each cell obtains data from its two neighbors, and its outputs are sent to these neighbors for the operation in the next cycle. Hence, any transmission delay would increase the time interval between fetching two adjacent samples. This increased time interval implies a lower sampling rate. However, because the communication links are localized and the number of links does not increase with the filter order, we can assign a dedicated high bandwidth link for each message transmission. Therefore, the transmission delay could be negligible compared to the execution time of each

cell. Thus, the overall sampling rate should remain essentially unchanged even considering the transmission delay.

#### 4.2.3. SSIMD Mode

The transmission delay of messages will seriously degrade the performance of Barnwell's structure. Unfortunately, due to its extremely complex interconnection (See Figure 2-6b), this transmission delay seems inevitable. Furthermore, as described in section 2.4, the highest throughput depends on the availability of all its delayed recursive inputs, which are usually generated in some other PE's. Therefore, a delay on any link would delay the available time of the recursive inputs of some other PE's. We will give a detailed discussion on the effect of this delay time on the sampling rate.

Barnwell has pointed out[9] that the largest potential problem in SSIMD solutions concerns the inter-PE communication. However, he also claimed that the fundamental periodicity of the SSIMD solution makes the communications requirements very uniform, which avoids many potential time conflicts. He also says that the nature of the communications environment can be systematically controlled by a long delay chain. The first claim is true, however, the second is true only if shared memories is allowed. Owing to the finite speed of memory access, the memory can only be shared among a limited number of PE's. For very high rate filters, it is more feasible to transfer data among PE's rather than to store them in a common memory space. Then, the extremely complex data flow pattern will surely cause a serious problem in achieving high sampling rate in a multiprocessing system.

The delay effect can be analyzed by looking at Figure 4-1. If there is a delay time "d" for transmitting the recursive output of PE 0 to PE  $l$ , the earliest time that PE  $l$  can start computing is also delayed by the same amount. Hence, the skew time between any two adjacent PE's becomes  $\frac{R-l(l)+d}{l}$ . The maximum

number of PE's that can be used without introducing any idle time becomes

$$M_z = \text{INT}[M(l_z)] = \text{INT}\left[\text{MIN}_{i,l}\left[\frac{lT}{R - I(l) + d_{li}}\right]\right] \quad (4-4)$$

where  $d_{li}$  is the delay time for the  $i^{\text{th}}$  delayed recursive input on PE  $l$ . Since the throughput rate is inversely proportional to this number, the effect of this delay on the throughput rate is easily seen. For high order filters, since more communication links are required for each PE, the throughput rate will be greatly slowed down, if there is a huge delay on any communication link to any PE.

One way to ease this problem is to apply this structure to each subfilter of the cascade or parallel form. Within each subfilter, only two incoming links for the recursive inputs are required and the delay time between cascading stages does not affect the overall throughput. Hence, it can achieve the same sampling rate while suffering much less from the communication delay. However, it is impossible to totally eliminate the delay effect, unless only one PE is used.

A side effect of this complex interconnection is that since the number of pins on each chip is limited, some dedicated switching PE's might be needed to handle this complex data transfer. Therefore, the actual number of PE's may be correspondingly more than expected. These switching PE's are not required in some simple structures, such as the other three SISO filter structures. Thus, this extra usage of PE's should also be counted when comparing its performance with other structures.

#### 4.2.4. Block I/O Filters

As mentioned earlier in this section, only the transmission delay in the feedback operations will affect the overall throughput rate. Hence, for the block I/O filter, only the feedback product matrix  $B_0^{-1}B_1$  has to be considered. The realization of the other two product matrices is not essential as far as the throughput rate is concerned.



The delay effect on the sampling rate for the block I/O filters depends on how the operations of the feedback matrix are scheduled. If any communication links exist for connecting the PE's computing the feedback submatrix, transmission delay on these links will degrade the speed performance. This delay time may introduce some idle time in the PE's, and hence increase the actual execution time of the affected PE's. The block size is then determined by the PE which has the longest execution time. Fortunately, the size of the feedback matrix does not change with the input block length. Hence, the communication environment will not be more complicated, even if the block size has to increase to compensate this transmission delay.

If multiprocessing is employed to compute the feedback submatrix, a broadcasting type of communication is required. Hence, the transmission delay is likely to occur in such a complex communication environment. Assigning each row to one PE, as mentioned in section 4.1.4, is not a good scheduling as far as the delay effect is concerned. One way to avoid this delay effect, of course, is to assign the  $N \times N$  lower right submatrix of  $B_0^{-1}B_1$  to one PE. Then, the feedback part is contained within a single PE, and hence the sampling rate is not sensitive to any transmission delay. However, because of the longer execution time, the filter latency is much longer (especially for high order filters).

#### 4.2.5. Block State Implementation

In the case of block diagonal state matrix, if each submatrix on the diagonal is assigned to one PE, the transmission delay has no effect on the throughput rate, for no communication is required for the feedback matrix. Therefore, throughput rate stays the same even if there is a huge delay on any data transfer path. For the systolic approach, since data flows in a regular and simple manner, the data transmission can be very efficient. As distinct from the systolic array filter described by Kung, only one link is required between any two

adjacent cells and the data transmission is unidirectional. Hence, more pins can be used for one message transmission. This makes the data transfer even more efficient.

For Zeman's structures, the communication environment is similar to the parallel and cascade forms and hence the throughput is insensitive to the communication delay.

#### 4.3. Latency Consideration

As mentioned in section 3.2, latency is sometimes an important factor that limits the applicability of digital filters to real time signal processing. Although most signal processing or communication systems can tolerate latency in some applications, such as real time speech or where a filter is in a feedback loop, latency is important. It is usually desired that this delay be as short as possible. Most importantly, latency is very susceptible to the message transmission delay. Since latency includes both the execution time of the PE's and the transmission delay time, the latency of a structure whose throughput is susceptible to the transmission delay is also sensitive to this delay.

##### 4.3.1. SISO Filters

For the cascade form, the filter latency is the sum of the execution time of all the cascading stages if there is no transmission delay. Thus, this latency is similar to that of a direct form realization. However, since the constant term is distributed into all the stages, the total execution time is usually larger for the cascade form than the direct form. Besides this, the transmission delay on each link adds to the overall latency.

The parallel form has much smaller latency than the cascade form. The latency equals to the execution time of a second order filter plus the processing time of the final summing node as well as the transmission delay on the links

from the filter sections to this summing node. In the worst case, the highest transmission delay among all the links connecting all the filter sections to the summing node should be added to the overall latency. Thus, the delay effect on the latency is not so severe as in the cascade form.

As for the systolic array realization, since each output sample goes through all the cells before it can be sent out, the overall latency equals that of the direct form. As discussed in section 4.2.2, the transmission delay should be negligible for this implementation.

The latency of Barnwell's structure is also equivalent to that of a direct form realization, if the number of PE's does not exceed  $M_2 + 1$ .

#### 4.3.2. Block Filters

For block state filters, latency is extremely important, since it is the only limiting factor. For block filters, the overall latency is also governed by its structure. Since the two dimensional systolic array structure for the block state filter is the most attractive structure, its latency behavior deserves further investigation. A problem that was left open in section 3.4 is the way of grouping four cells into one PE. We give here a detailed discussion about how this cell grouping can affect the latency.

From Figure 3-5, it is clear that the input data can go through either matrices B and A or matrix D to get to the entries of matrix C. Since the state variables can be precomputed in the previous time interval, The overall latency is the sum of the execution time of all the cells on the longest path from the input to the output through matrix D and matrix C. In other words, the maximum number of cells on those paths determines the latency and should be minimized. From the above argument, obviously, the best way to group the cells is to combine four cells on the same column in either matrix D or matrix C into

one. Since all the columns are operated in parallel, this combination would minimize the number of cells.

For perfect transmission (no transmission delay), the filter latency is simply the overall execution time on the path which requires the longest time. Therefore, for the linear systolic array and SSIMD approaches, the latency equals to that of a direct form implementation, which has  $2N$  multiplications, where  $N$  is the order of the filter. For the block state filters, the number of multiplication on the longest path, which passes through matrices  $D$  and  $C$ , is  $L+N$ , if four cells on the same column are grouped together. Since  $L$  is usually much larger than  $N$  for high sampling rate signals, this latency is longer than that of the SISO filters.

#### 4.4. Number of Operations

IIR filters are known to require lower computational rate than their equivalent FIR filters. Hence, unless linear phase is essential or there is some restriction precluding recursive operation such as adaptation, IIR filters are usually used to save chip area. However, as the block size increases for block filters, the average number of multiplications required to generate one output sample also increases. On the other hand, it is straightforward to increase the throughput rate of an FIR filter while keeping the number of multiplications per output sample unchanged. Hence, block IIR filters lose their advantage over FIR filters, at high sampling rates. This implies that FIR filters are better choices for processing signals sampled at extremely high rates.

Although the average number of operations for generating one output sample may not be important for a multiprocessing system, it does reflect the die area or the number of chips required. Thus, among the structures that can achieve the same desired throughput, it is advantageous to choose the structure with minimum number of multiplications. However, some other factors, such as

the complexity of implementation and interprocessor communications etc., should also be considered before deciding which structure to choose. In this section, we will concentrate on the comparison of computational rate between FIR and IIR block state filters.

Although the FIR filter structure as shown in Figure 1-2 with direct form implementation in each section can achieve any desired rate as a block state filter, the input data distribution pattern would be very complex. It is not fair to compare the number of operations between this structure and the block state structure as shown in Figure 3-7 in section 3.4. The communication overhead of Figure 1-2 might severely degrade the filter performance. Although the sampling rate may not be affected, the extra hardware for handling the data switching seems unavoidable. Hence, the number of operations itself does not fully reflect the hardware complexity. On the other hand, the block FIR filter implementation as shown in Figure 3-6b has a very similar structure to the block state filter. The computational rates of these two structures will be compared.

For an  $N^{\text{th}}$  order block state IIR filter with block diagonal state matrix, the total number of multiplications to generate an output vector is given as

$$2N + 2LN + \frac{L(L+1)}{2}$$

This number is obtained by assuming that all the poles are complex and distinct and by considering the fact that the matrix  $D$  is lower triangular. Therefore, the average number of multiplication to generate one output sample is

$$\frac{(L+1)}{2} + 2N + \frac{2N}{L} \quad (4-5)$$

Suppose an FIR filter with  $M$  multiplications per output sample has similar frequency response as the  $N^{\text{th}}$  order IIR filter. We should use an FIR filter instead of an IIR filter, if the block size  $L$  satisfies the following relationship.

$$\frac{(L+1)}{2} + 2N + \frac{2N}{L} \geq M$$

The value  $M$  depends on the actual implementation of the FIR filter and how we compare IIR and FIR filters. If the fact that computation might be saved due to the symmetry of FIR filter coefficients is neglected, this value is the number of taps.

The comparison between FIR and IIR filters is not simple Rabiner et. al.[35] made a comparison by looking at the passband and stopband ripples as well as the transition band. In Fig. 7b of their paper, a sixth order elliptic filter is approximately equivalent to a 41-tap FIR filter. Plugging these numbers into equation (4-4), an FIR filter should be used if  $L$  exceeds 56 (See Figure 4-3). This number depends heavily on the criteria for comparison and the implementation of the FIR filter. So, this upper bound of  $L$  should be computed for each filter design.

If a typical existing signal processor is chosen as a PE to realize the IIR block state filter with a two dimensional systolic array structure, the sampling rate with a block size at the cross point in Figure 4-3 can be estimated to be high enough for most applications. Suppose the internal clock rate of each PE is 5MHz and the I/O overhead can be ignored. The block state filter structure can generate a block of output samples (56 samples in this case) in the time of performing four multiplications and four additions. If the PE's can perform accumulation (one multiplication and one addition) in one clock cycle, the vector throughput rate will be 1.25MHz. The scalar sampling rate can then be easily calculated to be  $1.25 \times 56 = 70\text{MHz}$ . In most practical applications, this sampling rate is higher than necessary. In other words, the IIR block state filters should be adopted for most applications.

An obvious fact that can be inferred from Figure 4-3 is that the average number of multiplications decreases initially as the block size increases. The number of multiplications per output sample reaches a minimum point for a

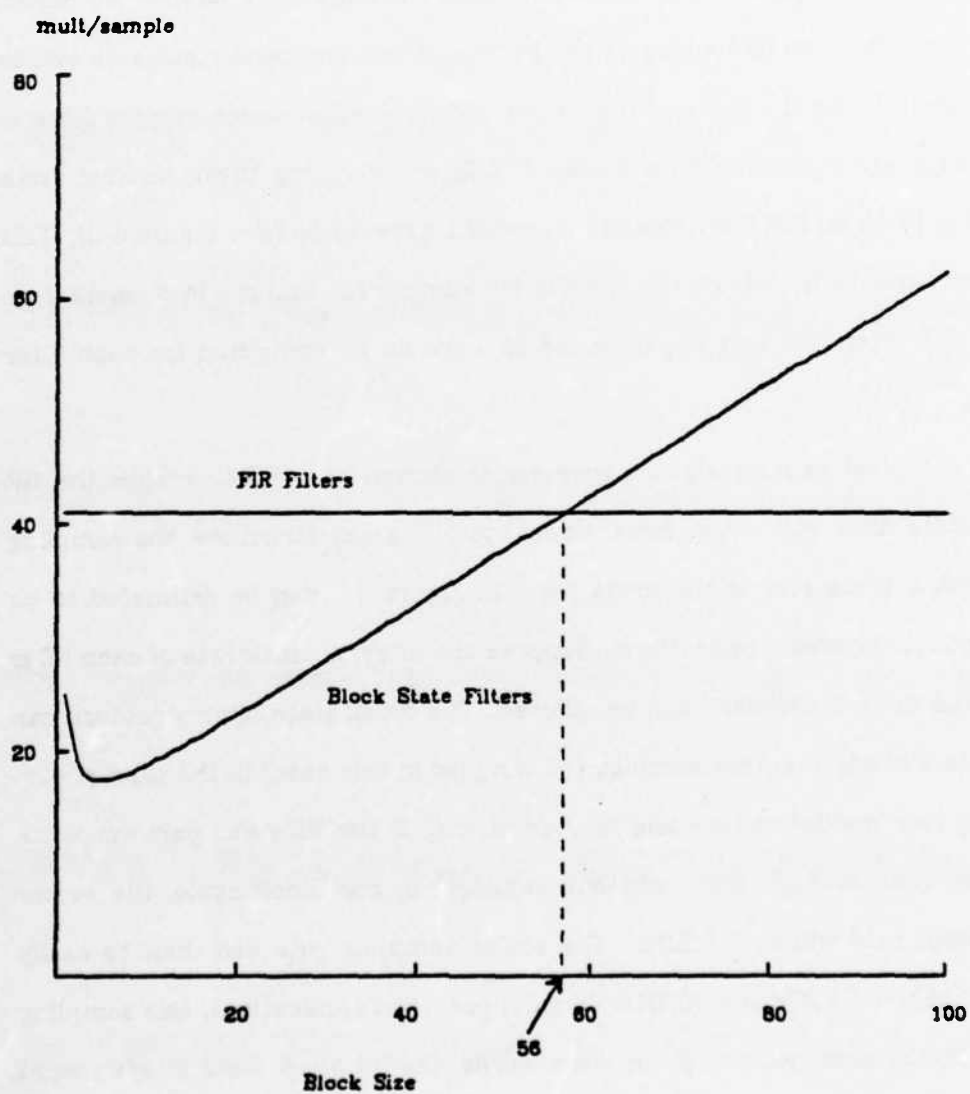


Figure 4-3 Computational Rate of FIR and IIR Block Filters



certain block size, which is greater than 1. This is similar to Zeman's conclusion on the optimal block length. Thus, even considering serial processing only in state space design, buffering input samples for some size can achieve higher throughput rate than the corresponding SISO filter.

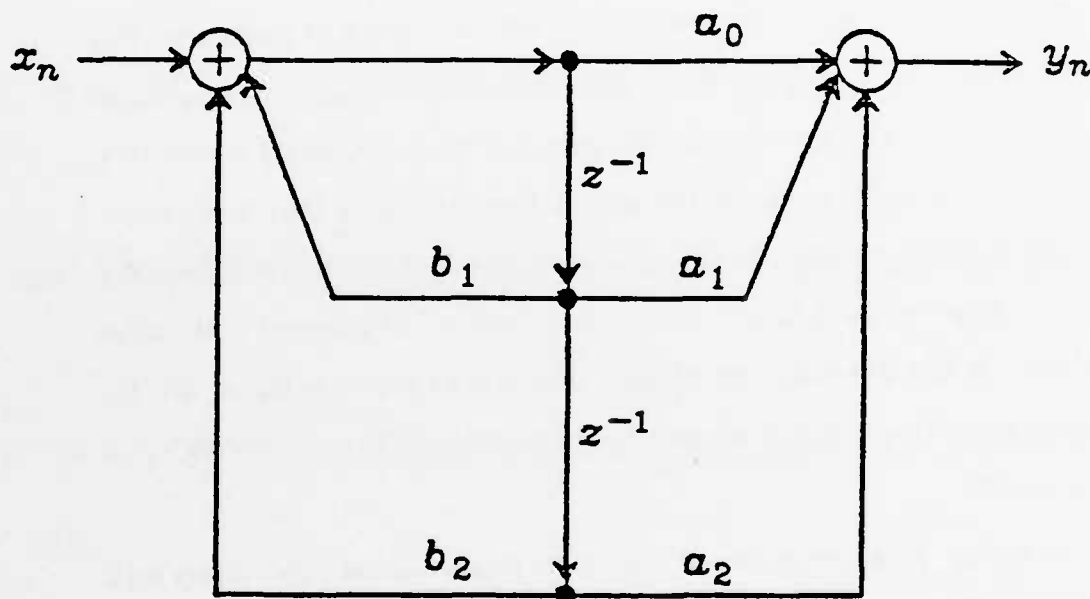
#### 4.5. Hardware Usage

SISO filters can easily achieve a highly efficient usage of hardware. This high efficiency is inherent in the algorithms. All the four SISO structures described in chapter 2 have exactly the same operation for almost all the PE's. Barnwell's structure is perfect in this aspect, because it is an SIMD mode structure. All the PE's execute the same code but on different input samples. The hardware usage for block filters, on the other hand, is not inherent in the filter equations. In the tree structure of inner products as shown in Figure 3-1, for instance, is not easy to find a perfect scheduling such that all the PE's are evenly loaded.

The systolic array realization of block state filters can not only achieve high throughput with localized communication but also be verified as near optimum. It is optimal in the sense that it requires the least number of PE's to achieve the same throughput compared to any other structures. Intuitively, since each cell in Figure 3-7 contains exactly the same number of operations, no idle time is required for any PE to wait for the data from any other PE's. However, since it is unlikely that the total number of entries in each of the three big rectangles in Figure 3-7 is a multiple of four, some cells must have less number of operations. This is why this structure is called near optimum.

Assuming perfect division, the optimality can be easily proven by Renfor's theorem[36]. Before showing this theorem, some important terminology is summarized as follows.

- (1) A directed graph  $N$  is a graph with directed links between nodes. The label on each link represents either a multiplying constant or a unit sample delay. Figure 4-4 shows the directed graph of a direct form realization of a second order filter.



**Figure 4-4 Directed Graph of a Second Order Filter**

- (2) The *arithmetic delay*  $d_{\text{ar}}$  is simply the time to perform an arithmetic operation. The time for an addition is associated with the link that comes out of the adder.

For the  $l^{th}$  loop, define

$$D_1 = \sum_{i \in I} d_{1i}$$

as the total time for all the arithmetic operations in this loop. In order that this loop is computable, the following relation must satisfy:

$$D_i \leq n_i T \quad (4-6)$$

where  $T$  is the unit sample delay and  $n_i$  is the number of unit delays in this loop.

The above equation must hold for all the loops in the network. Let's choose

$$T_0 = \max_i \{D_i / n_i\} \quad (4-7)$$

where the maximum is over all the directed loops. A loop in which this maximum is reached is called a *critical loop*. Now the theorem can be stated as follows.

*Theorem 4-1*

A digital filter network with unit delay  $T$  is computable if and only if  $T \geq T_0$ , where  $T_0$  is defined by (4-6).

For the block state filters, since any sampling rate is achievable, the maximum achievable sampling rate described in the above theorem is meaningless in this case. Therefore, the optimality issue should be addressed in a different manner. In the following discussion, we define the optimal network as a structure which can achieve a given sampling rate with the minimum number of PE's. Assume that the filter has only simple order poles and the state matrix is in a block diagonal form.

Since there is only one delay element in the block state filter, the highest sampling rate can be decided by looking at the feedback operation by Renfor's theorem. Since this operation also determines the block sizes, the comparison between different numbers of PE's for different block sizes is equivalent to the comparison between their average number of operations used to achieve some fixed sampling rate. From Figure 4-3, it is clear that an optimal block size exists which is always larger than 1. Actually, from section 4.1.5 it is clear that this number is  $L_{opt} = 2\sqrt{N}$ .

From section 3.2, it is clear that grouping one submatrix into one PE can give us a simple interconnection with the smallest block size. Suppose this

block size is  $L_0$ . Since grouping multiple submatrices into one PE also results in simple interconnection among PE's, the best choice of the block size should be a multiple of  $L_0$ , which is closest to  $L_{opt}$ . For most applications,  $L_{opt}$  is small enough and hence  $L_0$  is usually the best block size to choose. However, if a larger block size is necessary for the optimization, the accompanied longer latency should also be considered when choosing the block size.

#### 4.6. Conclusions

The comparison of performance among all the filter structures mentioned in chapters 2 and 3 has been discussed in this chapter. It appears that with conventional single-input single-output systems, an upper bound on the sampling rate always exists. Among the SISO filters, Barnwell's structure can achieve the highest sampling rate if the same hardware is used. On the other hand, the block processing of the input samples can remove this limitation. Higher throughput rate can be achieved with larger input block size. Hence, when high throughput rate is desired, block filters rather than SISO filters should be used.

When circuit size gets large, the data transmission within a chip or among chips might take more time and may affect the speed performance. Structures such as systolic arrays require only local and regular interconnections and hence data transmission paths can be short and the transmission time can be short. Furthermore, this local interconnection is also advantageous for VLSI design, since the interconnection wires are expensive in VLSI circuits. Therefore, a block state filter with systolic arrays is the most efficient structure as measured by the throughput and the transmission cost. Barnwell's structure suffers more from the data transmission delay due to its complex interconnection pattern. Hence, although it can achieve higher sampling rate than the other SISO filters, the transmission delay might make it inferior.

The comparison between the block state filters and their equivalent FIR filters is also presented. It is clear from section 3.1 that the two dimensional systolic array structure can also be applied to the implementation of FIR filters. Hence, the FIR filter can be realized with only local interconnection among processing elements. The throughput rate of the FIR filter can be increased by increasing the dimension of this array in a similar fashion as the block filters. However, the average computation of an FIR filter does not increase with the block size, whereas for block filters, this rate does increase with the block size. This rate affects the requirement on the number of PE's and hence should be minimized. The comparison between the rates of FIR and block filters shows that for very high sampling rate filters, the FIR filter is a better choice in terms of the hardware size.

## CHAPTER 5

### EFFECTS OF FINITE REGISTER LENGTH

From the discussion in the previous chapters, that block state filters outperform all the other parallel structures in overall throughput rate. However, another critical issue, which has not been discussed yet but would affect the complexity of the actual filter implementation, is the effect of finite register length. A structure which is sensitive to this effect requires more bits for representing each samples, and hence would slow down the arithmetic operations. Although these lower speed arithmetic operations do not prevent us from achieving high throughput rate if the block state structure is used, they do increase the required value of  $L$  and hence the amount of hardware. Therefore, the finite word length behavior of the block state filters will be studied in detail in this chapter.

#### 5.1. Introduction

When a digital filter is implemented with either a dedicated digital hardware or as a computer program, its coefficients and the input and output samples have to be quantized. This finite precision of filter coefficients and sample values will cause some error in the output signal. The sources of this quantization error are summarized as follows[37].

- (1) The filter coefficients must be quantized to some finite number of bits.
- (2) The input samples to the filter must also be quantized to a finite number of bits.
- (3) The products of the multiplications within the filter must usually be rounded or truncated to a smaller number of bits prior to subsequent operations, since each multiplication doubles the number of bits for full

precision.

- (4) When floating-point arithmetic is used, rounding or truncation must usually be performed before or after additions as well.

The complex arithmetic units also imply lower speed for each operation. Therefore, the finite register length effect is an important issue to characterize for each filter structure.

The first error source will be briefly treated in the next section and the third and the fourth error sources, which are usually called roundoff noise, will be discussed afterwards. As for the second error source, since it does not concern the filter structure and there is no way to minimize this error by optimizing the filter structure, it will not be discussed here. The discussion of this error which is usually referred to as quantization noise, can be found in most communication books. The third and fourth errors are different from the second error source in two respects

- (i) The data to be quantized is already digital in form, and
- (ii) the rounding or truncation of the data takes place at various positions within the filter, not just at its input.

Because of (ii), the roundoff error is potentially much larger than the input quantization error, and is one of the principal factors which determine the complexity of the filter implementation.

The exact analysis of the roundoff error behavior is extremely difficult since the generation of this error is a highly nonlinear process. It is helpful to perform an approximate analysis by representing the effect of rounding in terms of an additive error signal, which is referred to as roundoff noise. The filter can then be modeled as a linear filter associated with noise sources at various places. The output noise statistics can then be analyzed by standard random



process techniques.

For simplicity, only fixed point arithmetic, where rounding occurs only in multiplication but not in addition, will be considered. For two's complement arithmetic, it is usually assumed that the roundoff noise of a single rounding can be modeled as a white noise with a variance given by [1, Chap. 9].

$$\sigma_0^2 = \frac{\Delta^2}{12} \quad (5-1)$$

where  $\Delta$  is the quantization step size. With fixed point arithmetic, this step size is uniform over the whole dynamic range and hence a constant power noise source is associated with each rounding error. For an  $n$ -bit register length, this step size is  $2^{-n}$ , if the dynamic range is unity. When multiplying two  $n$ -bit numbers, a  $2n$  bit register is required to keep the accuracy of the product. This outcome must be rounded to  $n$ -bit, if it is fed back as an input for the calculation in the next cycle. If this outcome is not fed back in the next cycle, a wider register can be used to keep its full precision. This is why recursive filters are potentially noisier than nonrecursive filters. Thus, an error results with a mean square value  $\frac{2^{-2n}}{12}$ . The equivalent filter considering the roundoff noise of the original direct form second order filter is shown in Figure 5-1. In this graph, the rounding is assumed to occur after each multiplication and before the summation.

The effect of the first error source will be discussed in the next section. Emphasis will be put on the stability problem, which results from the pole movement due to the finite precision of the filter coefficients. Block state filters will be shown to be good structure in this aspect. The roundoff noise behavior will be discussed in the following sections. A detailed derivation for filters in the state space design will be presented to verify that block state filters have less roundoff noise as the block size increases.

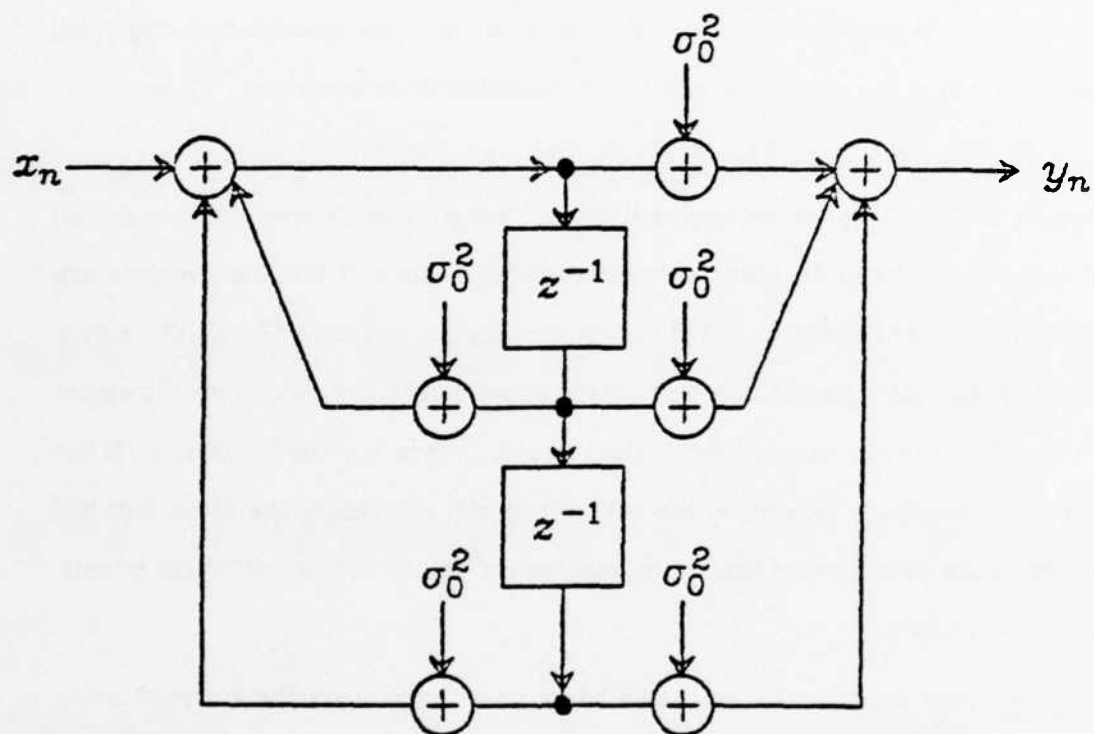


Figure 5-1 2<sup>nd</sup> Order IIR Filter with Roundoff Noise

### 5.2. Stability Consideration

The first error source usually results in some deviation in the actual frequency response from the desired one. Hence, the implemented filter has a frequency response which is not exactly the same as with infinite precision coefficients. An even more serious problem resulting from the finite precision of the filter coefficients is the stability problem. Quantization of the filter coefficients to finite length can cause a shift of the pole locations. If a filter has a pole close to the unit circle with infinite precision, this quantization may push this pole out of the unit circle and hence make the filter unstable.

For the block state filter realization, the filter stability becomes less sensitive to coefficient quantization, as the block length  $L$  increases. For a minimal realization in the state space domain, the eigenvalues of the state matrix are equivalent to the poles of the filter. According to equation (2-32), the eigenvalues of the state matrix of a block state equation are the corresponding eigenvalues in the scalar case raised to the power  $L$ , where  $L$  is the block size. If the poles are originally located within the unit circle, increasing the block size will move all the poles toward the origin, making instability due to coefficient quantization less likely.

This pole dependence on  $L$  also has a great impact on the roundoff noise behavior, as will be discussed in detail later in this chapter. However, intuitively, a smaller eigenvalue implies a shorter time constant. Hence, the noise sources generated on the internal nodes decay faster in time in the block case than in the scalar case. Suppose a filter has a single eigenvalue  $\lambda$  and an initial state  $\alpha_0$  at time 0, then this initial state will contribute to the output by an amount  $\alpha_0 \lambda^n$  at time  $n$ . This can be easily seen in Figure 5-2, which is a signal flow graph of a single pole filter. For a fixed initial state, as time goes on, the contribution of this state to the output decays faster for a smaller eigenvalue

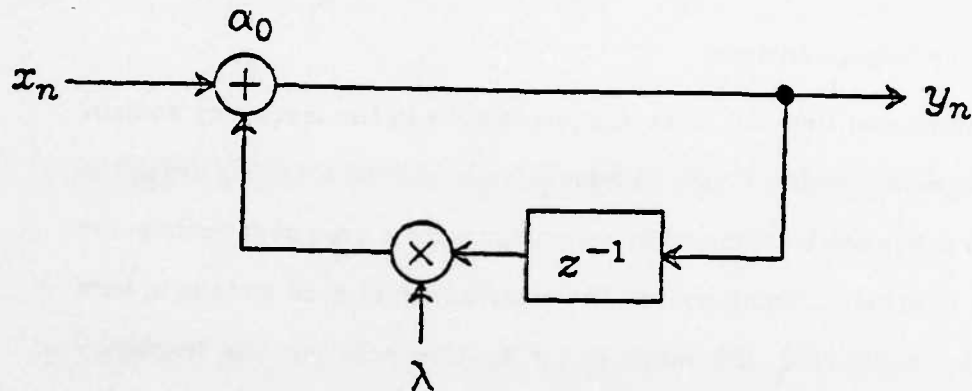


Figure 5-2 Signal Flow Graph of a Single Pole Filter

than for a larger eigenvalue. If at time  $m$  ( $m \leq n$ ), a noise is generated at the summing node with power  $\alpha_0$ , the overall output noise power at time  $n$  is the sum of all the output noise powers contributed from the noise sources generated before time  $n$ . This overall noise power at time  $n$  can be represented by

$$\sigma_n^2 = \sum_{m=0}^n \alpha_0 \lambda^m$$

Obviously, for smaller eigenvalues, or equivalently smaller pole magnitudes, the overall output noise power at time  $n$  is less than for larger eigenvalues. In conventional filter design, the filter response will be totally different if the pole position is shifted. On the other hand, the block state filter can achieve the same scalar transfer function while reducing the magnitudes of all the poles; hence, the roundoff noise is reduced as the block length increases.

In the next section, the general approach for the roundoff noise analysis will be briefly outlined. This analysis, which is done in the frequency domain, is a very common technique. In the following sections, the roundoff noise analysis in the time domain will be discussed in detail by using the state space representations. The average noise power can be represented by the filter equation

directly, and no transfer function in the frequency domain is involved.

### 5.3. Frequency Domain Analysis

The conventional roundoff noise analysis is done in the frequency domain. Assuming white, uncorrelated noise sources at each internal summing node, the output noise power can be obtained by summing up the noise power contributed from all the internal summing nodes. The contribution of each node is in turn obtainable by multiplying the noise power at this node by the frequency response from this node to the output. Jackson[5] introduced a state variable approach to characterize the noise behavior.

A digital filter network may be represented as shown in Figure 5-3. The transfer function from the input to the  $i^{\text{th}}$  branch node is denoted as  $F_i(z)$ , and  $G_i(z)$  denotes the transfer function from this summing node to the output.  $H(z)$  is the transfer function from the input to the output. Associated with each summing node is the roundoff error generated at time  $n$ , which is labeled  $e_i(n)$ . This error has a mean square value  $k_i \sigma_0^2$ , where  $k_i$  is the number of products to be summed up at this summing node. If rounding is performed after summing up all the products, this number would be unity. The noise power generated at the output summing node is a constant and usually is negligible compared to the noise generated from the internal summing node.

Suppose this network is properly scaled; hence, the power gain from the input to each internal summing node satisfies some scaling rule. Then, the contribution of the  $i^{\text{th}}$  noise source to the overall noise power can be written as

$$k_i |G_i(e^{j\omega})|^2 \sigma_0^2$$

Since the noise sources are uncorrelated from source to source and from time to time, the output noise has a power spectrum

$$N(\omega) = \sigma_0^2 \sum_i k_i |G_i(e^{j\omega})|^2$$

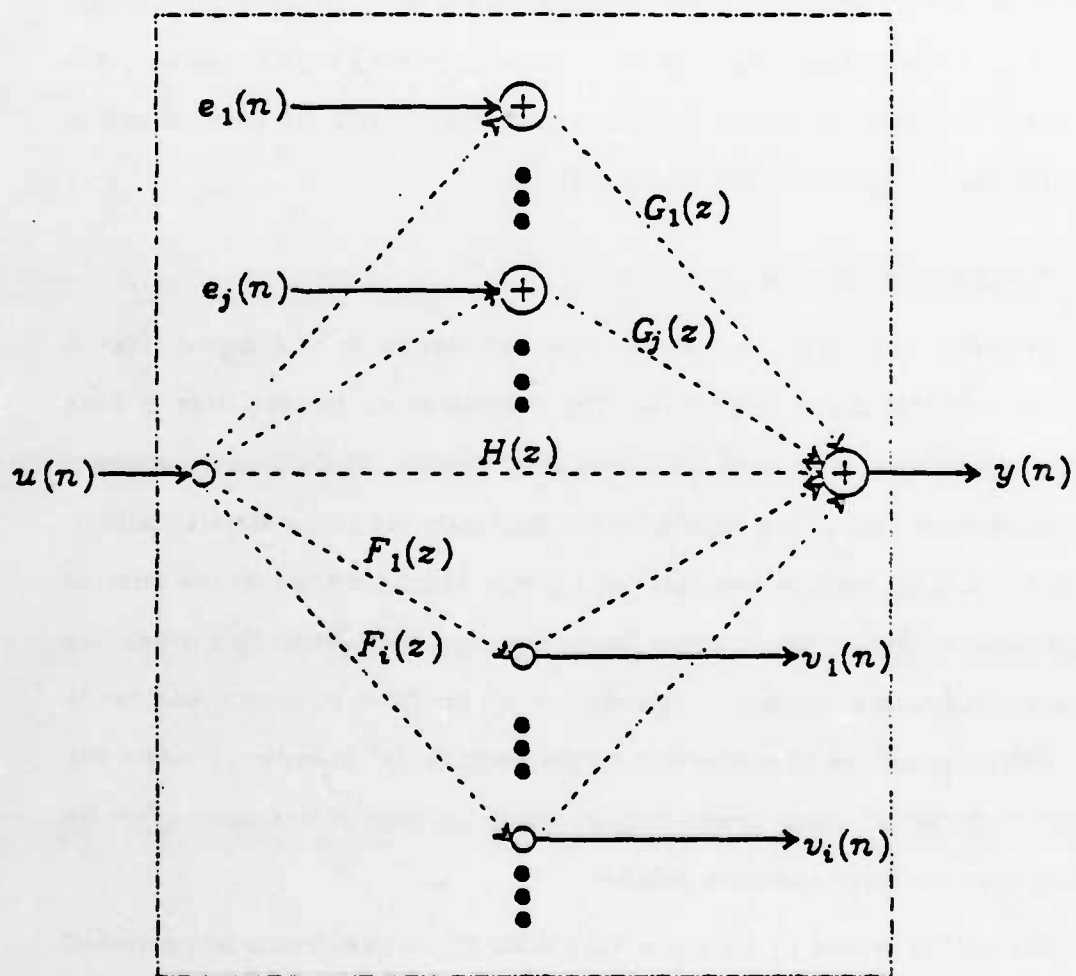


Figure 5-3 State Variable Representation of a Digital Filter

The variance, or total average power, of the output noise can then be obtained by the following equation

$$\sigma_y^2 = \frac{1}{\omega_s} \int_0^{\omega_s} N(\omega) d\omega$$

where  $\omega_s$  is the radian sampling frequency given by  $\omega_s = 2\pi/T$ . From the above derivation, it is clear that in order to obtain the overall noise power the calculation of the transfer functions from each internal summing node to the output is required. This tedious calculation makes the noise analysis and the comparison between various structures extremely difficult.

#### 5.4. Time Domain Analysis

As mentioned before, the state space representation of a digital filter is very desirable for the roundoff noise. The analysis, as will be seen later, is done in the time domain instead of in the frequency domain. The output noise power can be represented, at any time  $n$ , by the four matrices in the state equations. Thus, there is no need to calculate the transfer functions from all the internal state nodes to the output as in the frequency domain analysis. This is possible because the state equations completely specify the filter structure, and hence roundoff noise can be characterized by the coefficients. In order to obtain the steady state output noise power, simply look at the limit of this noise power by letting the time index approach infinity.

Barnes[38] proved in his paper that MIMO filters have much less roundoff noise than their corresponding SISO filters in the state space design. Actually, the noise power generated from the recursive operation for an SISO filter is equivalent to the sum of the noise power of all the samples in a block for the corresponding MIMO filter. Hence, the average noise power over one block outputs is decreased as the block size  $L$  increases. However, all his conclusions are based on the assumption that the rounding occurs, at each state node, after



summing up all the products. Thus, each summing node has a noise source whose noise power equals that in equation (5-1). A wider adder is then needed at each summing node to achieve this performance. Zeman[23] showed some results which verified Barnes' conclusions.

Even if each product is rounded before sent to be summed up, the average noise power generated from the recursive operation is still less than in the corresponding SISO filter. Hence, even with a narrow adder at each summing node, the roundoff noise can still be reduced compared to an SISO filter design. However, the noise power generated from the output summing node increases linearly with  $L$ . If this noise term becomes dominating, a wider adder is suggested.

The coefficients of an MIMO filter can be expressed by the coefficients of its corresponding SISO filter (See equation 2-32). As will be seen later, MIMO and SISO filters have similar formulation for the output noise power. Hence, minimizing the noise power for the SISO filter would also minimize the noise power for MIMO filters. Therefore, the optimum block state filter design becomes an SISO state filter design.

The roundoff noise analysis in the state space structure, has been investigated by various researchers[39, 40, 41]. The necessary and sufficient conditions for an optimal state space filter design have been developed. Mullis and Roberts proposed a block optimal filter structure which has near optimal noise performance with much less computation. Instead of designing an optimal filter directly, they independently optimized each section of a parallel and/or cascade connection of low order subfilters. Jackson et al.[17], mentioned another structure which is called sectional optimum. Instead of optimizing the whole filter in a block form, they optimize each  $2^{nd}$  order section independently. This structure requires a very simple design procedure and has noise performance close

to that of the block optimal structure. Above all, this structure can also have the efficient structure described in chapter 3. Therefore, it is possible to realize a high speed filter with localized communication while lowering the roundoff noise.

The sectional optimal filter design problem is reduced to an optimal second order filter design. Jackson et al.[17] developed a procedure for a minimum noise second order state space filter design. Barnes[20] introduced another special structure called normal filters which have the uniform-grid structure of Rader and Gold[42] and will not support autonomous overflow limit cycles[18]. Hence, this structure has a transfer function which is less sensitive to the coefficient quantization error and the pole positions. This is important for block state filters, for its poles are more widely spread and closer to the origin.

In this section, the roundoff noise formulation for both SISO and MIMO filters will be derived in the state space design. It will be clear that block filters do improve the roundoff noise behavior, once the noise representation is developed. The interaction between roundoff noise and dynamic range will be treated in a later section. The effect of scaling on the roundoff noise will also be discussed. With a fixed scaling rule, the conditions for an optimal filter design will be given and the two structures for a low roundoff noise second order state space filter design mentioned above will be derived.

#### 5.4.1. Noise Formulation for SISO Filters

The state equation for an  $N^{\text{th}}$  order digital filter is rewritten as follows

$$r_{n+1} = Ar_n + bx_n \quad (5-2a)$$

$$y_n = cr_n + dx_n \quad (5-2b)$$

where A, b, c and d are respectively,  $N \times N$ ,  $N \times 1$ ,  $1 \times N$  and  $1 \times 1$  real constant matrices. Due to the product quantization, the actual filter implemented by a finite word length machine is

$$r_{n+1} = Ar_n + bz_n + a_n \quad (5-3a)$$

$$y_n = cr_n + dz_n + \beta_n \quad (5-3b)$$

where  $a_n$  is the noise vector generated from the product quantization of  $A$  and  $b$  at time  $n$ , and  $\beta_n$  is a scalar noise term generated from  $c$  and  $d$ . Subtracting (5-2) from (5-3), we have

$$\Delta r_{n+1} = A\Delta r_n + a_n \quad (5-4a)$$

$$\Delta y_n = c\Delta r_n + \beta_n \quad (5-4b)$$

where  $\Delta r_n$  is the state-error vector  $r_n - r_n$ , and  $\Delta y_n$  is the output noise at time  $n$ .

The solution to (5-4) is

$$\Delta y_n = c \sum_{j=0}^{n-1} A^{n-j-1} a_j + \beta_n$$

if assume that  $\Delta r_0 = 0$ . Under the usual assumptions that the product quantization errors are white noise and are statistically independent from source to source, and from time to time, the crosscorrelation between  $a$ 's and  $\beta$ 's is zero. Therefore, the output noise power at time  $n$  is

$$\begin{aligned} \sigma_{y_n}^2 &= E(\Delta y_n^2) \\ &= \sum_{i=0}^n E[cA^{n-i-1}a_i (\sum_{j=0}^n cA^{n-j-1}a_j)^T] + E(\beta_n^2) \end{aligned} \quad (5-5)$$

Also since the autocorrelation between  $a_i$  and  $a_j$  is zero if  $i \neq j$ , the above equation can be written as

$$\sigma_{y_n}^2 = \left[ \sum_{i=0}^n cA^{n-i-1} E[a_i a_i^T] A^{n-i-1T} c^T \right] + E[\beta_n^2] \quad (5-6)$$

Since each rounding introduces a noise with expected power  $\sigma_0^2$ , both expectations in the above equation equal this noise power multiplied by the number of roundings associated with the corresponding summing node. Assuming that  $q_i$  products are summed up at the  $i^{\text{th}}$  state and  $\mu$  products are summed up at the output, the above equation can be written as

$$\sigma_{y_n}^2 = \left[ \sum_{i=0}^n cA^i Q (cA^i)^T + \mu \right] \sigma_0^2 \quad (5-7)$$

where

$$Q = \text{diag}(q_1, q_2, \dots, q_N)$$

Usually  $\mu$  is a constant and is much smaller than the other summation term, and hence will be ignored in the following analysis. For a balanced state equation where all states have the same number of products to sum up, the matrix  $Q$  in equation (5-7) can be replaced by a scalar  $q = q_i$ ,  $i=1,2,\dots,N$ . The steady state noise power is the noise in (5-7) when  $n \rightarrow \infty$ .

$$\sigma_y^2 = \lim_{n \rightarrow \infty} \sigma_{y_n}^2 = \left[ q \sum_{n=0}^{\infty} c A_n c A_n^T + \mu \right] \sigma_c^2 \quad (5-8)$$

#### 5.4.2. Block State Structures

Given a properly scaled realization  $(\hat{A}, B, C, D)$  in a block state form, the output noise vector can be obtained similar to equation (5-8).

$$\sigma_y^2 = \left[ I + \sum_{n=0}^{\infty} C \hat{A}^n (C \hat{A}^n)^T \right] \sigma_c^2 \quad (5-9)$$

where  $I$  is a column vector whose elements are all 1's. Assume only one rounding is performed at each summing node for every input block. If this block state realization is related to the single state realization  $(A, b, c, d)$  by equation (2-32), the  $i^{th}$  noise component can be represented as

$$\sigma_{y_i}^2 = \left( 1 + \sum_{n=0}^{\infty} c A^{Ln+i-1} A^{(Ln+i-1)T} c^T \right) \sigma_c^2 \quad i=1,2,\dots,L \quad (5-10)$$

where  $L$  is the block size. If define the average noise power as

$$\bar{\sigma}_y^2 = \frac{1}{L} \sum_{i=1}^L \sigma_{y_i}^2 \quad (5-11)$$

it is easily seen that the average noise power is

$$1 + \frac{1}{L} \sum_{n=0}^{\infty} c A^n (A^n)^T c^T \quad (5-12)$$

The noise power gain, which is the infinite sum in the above equation, is  $1/L$  of the noise power gain in the scalar case. It is obvious that the average noise power involves only the coefficients in its corresponding scalar state filter. Thus, minimizing the roundoff noise power in the scalar case would also minimize the roundoff noise power in the block case. In the next section, we are going

to discuss the optimal filter in the scalar case, considering also the scaling effect.

### 5.5. Optimal SISO Filter Synthesis

From equation (5-8), the steady state noise power consists of two terms. The first one is generated from the recursive operation, whereas the second one from the output summing node. The second term is a constant term and usually is much smaller than the first one, especially for a narrow band filter. Furthermore, for a given filter in the state space design, nothing can be done to reduce this term to improve the overall noise performance. However, for the first term, its value varies with the choice of states. Hence, the overall noise minimization is equivalent to the minimization of this term. In the following analysis, we will refer the overall noise to the first term only, and will find criteria for a minimum noise filter synthesis.

Due to the finite register length, the internal node of the filter might overflow if the input samples are too large. The scaling of filter coefficients is usually necessary in order to keep the registers from overflowing. However, this scaling usually also changes the behavior of the roundoff noise. Therefore, the scaling has to be considered before discussing the roundoff noise behavior. Before discussing the scaling effect, define two symmetric matrices as follows

$$K = AKA^T + bb^T = \sum_{k=0}^{\infty} (A^k b)(A^k b)^T \quad (5-13a)$$

$$N = A^T N A + c^T c = \sum_{k=0}^{\infty} (cA^k)^T (cA^k) \quad (5-13b)$$

Since the summation term in equation (5-6) is a scalar, it can be rewritten as

$$\sigma_y^2 = \lim_{n \rightarrow \infty} \sigma_{y_n}^2 = E \left[ \sum_{k=0}^{\infty} (cA^k \alpha_1)^T (cA^k \alpha_1) \right]$$

For a given state space structure, the autocorrelation of  $\alpha_1$  is invariant in time and with correlation matrix

$$E[\alpha_i \alpha_i^T] = Q \sigma_0^2$$

Hence, the above equation can be rewritten as

$$\begin{aligned} \sigma_y^2 &= E \left[ \alpha^T \sum_{i=0}^N (cA^i)^T (cA^i) \alpha \right] = E \left[ \alpha^T W \alpha \right] \\ &= \text{Tr}(WQ) = \sum_{i=1}^N q_i W_{ii} \sigma_0^2 \end{aligned} \quad (5-14)$$

$\text{Tr}(\cdot)$  is defined as the sum of the diagonal elements of its matrix argument.

From (5-14), it is obvious that the total noise depends on the diagonal elements of the matrix  $W$ . Now, the effect of scaling on these diagonal elements will be investigated. Before doing that, let's look at the effect of linear transformation on this matrix. From the definition of  $K$  and  $W$  and Theorem 2-1, it is obvious that any transformation  $T$  will map  $(K, W)$  to  $(K', W') = (T^{-1}KT^{-T}, T^T W T)$ . It is also easily seen that the diagonal element  $K_{ii}$ , is actually the  $l_2$  norm of the gain from the input to the  $i^{\text{th}}$  state. According to the conventional  $l_2$  norm scaling rule, the gain from the input to each internal summing node should be unity. This is equivalent to setting all the diagonal elements of matrix  $K$  to unity. Hence, the transformation matrix  $T$  should be a diagonal matrix with each element

$$T_{ii} = \sqrt{K_{ii}}$$

This matrix transforms the diagonal elements of  $W'$  to

$$W'_{ii} = W_{ii} T_{ii}^2 = W_{ii} K_{ii}$$

So, after scaling, the output noise power becomes

$$\sigma_y^2 = \sum_{i=1}^N q_i W_{ii} K_{ii} \sigma_0^2 \quad (5-15)$$

For the case of a full matrix  $A$ ,  $q_i = N+1$  for all  $i$ , and hence

$$\sigma_y^2 = (N+1) \sigma_0^2 \sum_{i=1}^N W_{ii} K_{ii} \quad (5-16)$$

Mullis and Roberts proved in their paper[40] a very important theorem which is stated as follows

**Theorem 5-1**

Let  $K$  and  $W$  be two  $n \times n$  real, symmetric, positive definite matrices. Then

$$\left[ \frac{1}{n} \sum_{i=1}^n K_{ii} W_{ii} \right]^N \geq M_a \quad (5-17)$$

where

$$M_a = \frac{1}{n} \sum_{i=1}^n \mu_i$$

and the numbers  $\{\mu_1^2, \dots, \mu_n^2\}$  are the eigenvalues of the product  $KW$ . In order for equality to hold, it is necessary and sufficient that

$C_1: D_0^{-1} K D_0^{-1} = D_c W D_c$  for some diagonal matrix  $D_c$ .

$C_2: K_{ii} W_{ii} = K_{jj} W_{jj}$  for all  $i, j$ .

The minimum output noise variance is therefore

$$\sigma_e^2 = N(N+1)M_a \sigma_c^2$$

and this minimum noise can be obtained when the equality of equation (5-17) holds or equivalently, when a diagonal matrix  $D_c$  can be found to satisfy conditions  $C_1$  and  $C_2$ .

In the next subsection, a minimum noise second order filter will be derived according to the above theorem. This filter design can be used to synthesize a high order filter which is section optimum. The normal filter formulation will be presented afterwards. Although this filter has slightly higher roundoff noise than the optimal filter, it has uniform sensitivity to the coefficient truncation over the entire  $z$ -plane and does not support autonomous limit cycles.

**5.5.1. Optimal Second Order Filter Synthesis**

From Theorem 5-1, the necessary and sufficient conditions for an optimal filter subject to the  $l_2$  norm scaling are

$$W = DKD \quad (5-18a)$$

$$\text{and} \quad K_{ii} W_{ii} = K_{jj} W_{jj} \quad \text{for all } i, j \quad (5-18b)$$

where  $D$  is a diagonal matrix. Since, the  $l_2$  norm scaling requires  $K_{ii} = 1$  for all  $i$ ,



(5-18b) then becomes

$$W_{ii} = W_{jj} = 1 \quad \text{for all } i, j$$

Since the diagonal elements of  $W$  are equivalent to the  $l_2$  norm of the gain from the corresponding internal summing nodes to the output, the optimal network is characterized by having equal noise contributions from each error source.

The above conditions are satisfied if, and only if,  $D=\rho I$ ; and thus, an alternate condition which is both necessary and sufficient for optimality is simply

$$W = \rho^2 K \quad (5-19)$$

In the second order case, (5-19) is not changed by writing it as

$$W = \rho^2 M K M \quad (5-20)$$

$$M = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

where

because both  $K$  and  $W$  are symmetrical matrices with equal diagonal elements.

From (5-13a,b) it is readily seen that (5-20) can be rewritten as

$$\begin{aligned} A^T W A + c^T c &= \rho^2 M (A K A^T + b^T b) M \\ &= \rho^2 M A K A^T M + \rho^2 M b b^T M \\ &= M A M^T M A^T M + M \rho b (M \rho b)^T \\ &= (M A M^T) W (M A M^T)^T + \rho M b (\rho M b)^T \end{aligned}$$

Hence, (5-20) is satisfied by a network of the form

$$A^T = M A M^T$$

and

$$c^T = \rho M b \quad (5-21)$$

For complex conjugate poles, (5-21) can always be satisfied with real-valued coefficient matrices.

Suppose the coefficients have the following form

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$c = \begin{bmatrix} c_1 & c_2 \end{bmatrix}$$

(5-21) states simply that

$$a_{11} = a_{22}$$

and

$$\frac{b_1}{b_2} = \frac{c_2}{b_1} \quad (5-22)$$

This network can be synthesized from an arbitrary realization  $(A, b, c, d)$  as follows. From (5-21) if the transpose of the optimal network is formed and the states  $\tau_1$  and  $\tau_2$  are interchanged, the resulting network is identical with the original optimal network except for scaling. However, if we form the network  $(MA^t M, Mc^t/2, b^t M, d/2)$  and place it in parallel with the network  $(A, b/2, c, d/2)$ , an overall network with the above property and the same transfer function can be produced. Therefore, the optimal network can be synthesized simply by merging these two parallel networks into a single network  $(\tilde{A}, \tilde{b}, c, d)$  and then scaling according to the  $l_2$  norm scaling rule.

Suppose the filter has a transfer function

$$H(z) = d + \frac{\gamma_2 z^{-2} + \gamma_1 z^{-1}}{\beta_2 z^{-2} + \beta_1 z^{-1} + 1}$$

and is implemented, for example by

$$A = \begin{bmatrix} -\beta_1 & -\beta_2 \\ 1 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ c = \begin{bmatrix} \gamma_1 & \gamma_2 \end{bmatrix}$$

The coefficients of  $(\tilde{A}, \tilde{b}, c, d)$  are then

$$\alpha_{11} = \alpha_{22} = -\frac{\beta_1}{2} \\ \delta_1 = \frac{1}{2}(1+\gamma_2) \quad \delta_2 = \frac{1}{2}\gamma_1 \\ c_1 = \frac{\gamma_1}{1+\gamma_2} \quad c_2 = 1 \\ \alpha_{12} = \gamma_1^{-2}(1+\gamma_2) \left[ \gamma_2 - \frac{1}{2}\beta_1\gamma_1 \pm \sqrt{\gamma_2^2 - \gamma_1\gamma_2\beta_1 + \gamma_1^2\beta_2} \right] \\ \alpha_{21} = (1+\gamma_2)^{-1} \left[ \gamma_2 - \frac{1}{2}\beta_1\gamma_1 \mp \sqrt{\gamma_2^2 - \gamma_1\gamma_2\beta_1 + \gamma_1^2\beta_2} \right]$$

The expression under the radical in  $\alpha_{12}$  and  $\alpha_{21}$  is always positive for complex-conjugate poles, and hence the coefficients in the above equations are all real valued. After scaling, the resulting second order network is optimal.

### 5.5.2. Normal Digital Filters

Barnes and Farn suggested a special structure in the state space design, namely a normal digital filter which has uniform sensitivity over the entire  $z$ -plane to the coefficient quantization and does not support the autonomous overflow limit cycles.[18]. Another advantage of this normal structures is that the expression noise is explicit, whereas, Mullis and Roberts and Hwang only provided a numerical approach for the solution of minimum noise. They also showed, within this context, a way to minimize the roundoff noise assuming simple poles only. A realization is called normal if and only if its state matrix satisfies the following relation

$$A \cdot A^* = A^* \cdot A$$

where  $A^*$  is the complex transpose of matrix  $A$ .

Suppose the filter to be implemented has a transfer function

$$H(z) = d + \alpha_1(z - \lambda_1)^{-1} + \alpha_2(z - \lambda_2)^{-1} \quad (5-23)$$

The filter can always be minimally realized in the state space domain as

$$\begin{aligned} r_{n+1} &= Ar_n + bz_n \\ y_n &= cr_n + dz_n \end{aligned}$$

where  $A$  is a  $2 \times 2$  matrix with eigenvalues  $\lambda_1$  and  $\lambda_2$ . It is well-known that normal matrices possess orthonormal eigenvectors  $\{\varphi_n\}$ , such that

$$A\varphi_n = \lambda_n\varphi_n \quad \text{where } n=1,2$$

and

$$\varphi_n^* \varphi_m = \delta_{nm}$$

Thus, we can expand  $A$ ,  $b$  and  $c$  in the form

$$\begin{aligned} A &= \sum_{n=1}^2 \lambda_n \varphi_n \varphi_n^* \\ b &= \sum_{n=1}^2 \beta_n \varphi_n \\ c &= \sum_{n=1}^2 \gamma_n \varphi_n^* \end{aligned} \quad (5-24)$$

where  $A^*$  is the conjugate transpose of the matrix  $A$ . In terms of the expansion coefficients, the resulting system transfer function is given by

$$H(z) = d + \beta_1 \gamma_1 (z - \lambda_1)^{-1} + \beta_2 \gamma_2 (z - \lambda_2)^{-1}$$

Compared to equation (5-23), it is required that

$$\beta_i \gamma_i = \alpha_i \quad i=1,2 \quad (5-25)$$

in order to obtain the correct transfer function. The choice of  $\beta$ 's and  $\gamma$ 's so that (5-25) is satisfied is dictated primarily by scaling and roundoff noise considerations.

The K matrix can now be formulated as

$$K = \sum_{n=0}^{\infty} A^n b A^{nT} b^T = \sum_{i,j=1}^2 \frac{\beta_i \beta_j^*}{1 - \lambda_i \lambda_j^*} \varphi_i \varphi_j^* \quad (5-26)$$

For a balanced scaling, it is required that

$$K_{11} = K_{22} = 1 \quad (5-27)$$

Suppose  $\lambda_1$  and  $\lambda_2$  are complex conjugate eigenvalues

$$\lambda_1 = \alpha + j\omega = \rho e^{j\phi}$$

$$\lambda_2 = \alpha - j\omega = \rho e^{-j\phi}$$

In order to make the system matrix A real, the eigenvectors  $\varphi_1$  and  $\varphi_2$  must satisfy

$$\varphi_2 = \varphi_1^*$$

Since they are orthonormal vectors, one choice of these vectors is of the form

$$\varphi_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ j \end{bmatrix} \quad \varphi_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -j \end{bmatrix}$$

Thus, the system matrix A, as determined by (5-24), has the form

$$A = \begin{bmatrix} \sigma & \omega \\ -\omega & \sigma \end{bmatrix} = \rho \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}$$

To make b a real vector,  $\beta_1$  and  $\beta_2$  must satisfy

$$\beta_2 = \beta_1^*$$

Replacing  $\beta_i = \varphi_i^T b$  in equation (5-26), the diagonal elements of K are

$$K_{11} = \frac{b_1^2 + b_2^2}{2(1-\rho^2)} + \frac{s}{2}$$

$$K_{22} = \frac{b_1^2 + b_2^2}{2(1-\rho^2)} - \frac{s}{2}$$

where 
$$s = \frac{(b_1^2 - b_2^2) - \rho^2 \{ (b_1^2 - b_2^2) \cos(2\psi) - 2b_1 b_2 \sin(2\psi) \}}{1 + \rho^4 - 2\rho^2 \cos(2\psi)}$$

In order to satisfy the scaling condition (5-27), we must have

$$s = 0 \quad (5-28a)$$

$$b_1^2 + b_2^2 = 2(1 - \rho^2) \quad (5-28b)$$

(5-28a) implies that

$$\frac{b_2}{b_1} = \frac{\rho^2 \sin(2\psi) \pm \{1 + \rho^4 - 2\rho^2 \cos(2\psi)\}^{1/2}}{1 - \rho^2 \cos(2\psi)} \quad (5-29)$$

(5-29) and (5-28b) determine both  $b_1$  and  $b_2$ . Since  $\varphi_i$ 's and  $b_i$ 's are known, the values of  $\beta_1$  and  $\beta_2$  are immediately available. Substitute  $\beta_i$ 's and (5-20) into (5-19), the vector  $c$  can be obtained as follows

$$c_1 = \frac{b_1(\alpha_1 + \alpha_2) + b_2(j\alpha_1 - j\alpha_2)}{2(1 - \rho^2)}$$

$$c_2 = \frac{b_1(j\alpha_1 - j\alpha_2) + b_2(\alpha_1 + \alpha_2)}{2(1 - \rho^2)}$$

Since  $\alpha_1 = \alpha_2^*$ ,  $c_1$  and  $c_2$  are real values.

### 5.5.3. Examples

Jackson, Lindgren and Kim[17] compared the roundoff noise performance among various structures. This comparison is based on the measurement of the noise power gain which is defined as the ratio of the output noise power generated from the recursive operations to a unit noise power which is defined as in equation (5-1). For SISO filters, this power gain is defined as the infinite summation term in equation (5-8). Table 5-1 shows the noise power gain for five different filters each of which is implemented in a parallel form. The second column shows the noise power gain if each parallel section is implemented by 2<sup>nd</sup> order canonical form. The normal and sectional optimal structures are shown in columns 3 and 4 respectively. In this parallel design, the block optimal is the same as the sectional optimal design.

Table 5-2 shows these five filters with cascade form designs. The block

Filter	Canonical	Normal	Sect.-Opt.
Chebyshev II BRF, N=6	13.8	11.0	10.9
Chebyshev I LPF, N=10	19.2	14.0	13.8
Elliptic LPF, N=10	16.8	13.7	13.5
Butterworth LPF, N=6 $f_c = 0.25 f_s$	14.2	14.6	13.4
Butterworth LPF, N=6 $f_c = 0.025 f_s$	27.0	14.8	13.4

Table 5-1 Noise Power Gain (in dB) for Parallel Form Designs

optimal form is not quite the same as the sectional optimal form and hence is also tabulated.

Filter	Canonical	Normal	Sect.-Opt.	Block-Opt.
Chebyshev II BRF, N=6	21.0	10.5	10.5	10.5
Chebyshev I LPF, N=10	29.9	24.2	24.5	24.1
Elliptic LPF, N=10	21.5	16.9	16.9	16.5
Butterworth LPF, N=6 $f_c = 0.25 f_s$	9.2	11.0	9.1	9.0
Butterworth LPF, N=6 $f_c = 0.025 f_s$	18.7	10.3	7.9	7.7

Table 5-2 Noise Power Gain (in dB) for Cascade Form Designs

From these two tables, the canonical form is worse than all the other forms in almost all the cases. On the other hand, normal, sectional optimal and block optimal are quite close to one another.

### 5.6. Conclusions

The effects of finite register length have been discussed in this chapter. These effects would affect the complexity of the filter structure and hence are critical factors in the actual filter implementation. Block state filters are proven to have better roundoff noise performance than their corresponding scalar state filters. Due to the similarity in the noise formulation between both cases, designing a minimum noise scalar state filter would automatically minimize the block state filters by using equation (2-32) to transform the scalar filter into a block filter. This relationship greatly simplifies the design procedure for designing a low noise block state filter. The section optimum structure can simplify the design process even further, since only the second order filter need be considered. A filter of any order can be easily synthesized from the second order filters. Two different approaches for low noise second order filter design are also derived.

Simulation results confirming the theoretical results here will be given in the next chapter.



## CHAPTER 6

## SIMULATION RESULTS

In this chapter, simulation results will be presented to verify the analysis of the previous chapters. The speed performance, including the delay effect and latency, of all the structures will be shown first. The roundoff behavior of the block state structure will then follow. The test filter is a six-order butterworth filter using bilinear transformation which has a transfer function as in equation (6-1)[1].

$$H(z) = \frac{0.0007378(1+z^{-1})^6}{(1-1.2686z^{-1}+0.7051z^{-2})(1-1.0105z^{-1}+0.3593z^{-2})} \cdot \frac{1}{(1-0.9044z^{-1}+0.2155z^{-2})} \quad (6-1)$$

The programs are written in 'C' and run on a VAX11-730.

## 6.1. Speed Simulation

A special C compiler 'simcc' is provided to work cooperatively with SIMON to obtain the execution time of each task. SIMON is a simulation program which executes the application programs as if they are running on a multiprocessing system. To ensure that no PE will receive any future data, which might be available for a simulator running on a single processor, SIMON has a global clock to keep track the elapse time of each PE. 'simcc' inserts instructions into the assembly language routine to accumulate the execution time of each instruction. If the programs of a process are compiled with the regular compiler 'cc', no execution time is counted. Therefore, this process can run virtually infinitely fast[2]. This is done for the simulation overhead, such as reading and writing data from and to the files. If compiling reading and writing with regular compiler, the speed limitation of the computer I/O will not affect the actual filter

speed simulation, since the input and output data are transmitted infinitely fast. In the following simulations, the execution time of all the instructions is assumed to be  $1 \mu s$ .

In this chapter, the simulation results for speed performance and transmission delay affect on both Barnwell's and the Block state structures will be shown. The cascade/parallel form and scalar systolic array approaches are straightforward. The structures are fixed with a given filter equation; hence, it is impossible to increase the throughput rate by changing the number of PE's. Block I/O filters are similar to block state filters but not so attractive as far as communication complexity is concerned. Therefore, these structures will not be discussed here.

#### 6.1.1. SED Filter

In this section, we show the speed performance of Barnwell's algorithm applied to the filter equation above with a varying number of PE's. The program structure is shown in Figure 6-1 for the case of 10 PE's. Processor 1 sends out 512 sample points and PE 12 collects all the output samples from PE's 2 to 11 and then puts them into the right order. Programs in PE's 1 and 12 are compiled with regular compiler; hence, they work as if they can finish the whole function in no time at all. Thus, the overall speed is limited by the filter function but not by the data input and output. The overall execution time is obtained by recording the arrival time of the first and the last sample in PE 12, and taking the difference as the overall execution time.

Figure 6-2 shows the speedup ratio vs. the number of PE's. This ratio is obtained by comparing the speed to that of only one PE. The straight line represents the ideal case where the speedup ratio increases linearly with the number of PE's with a unity slope. When the number of PE's is below 10, the speedup ratio increases linearly but with a lower slope. From 10 to 11, the rate

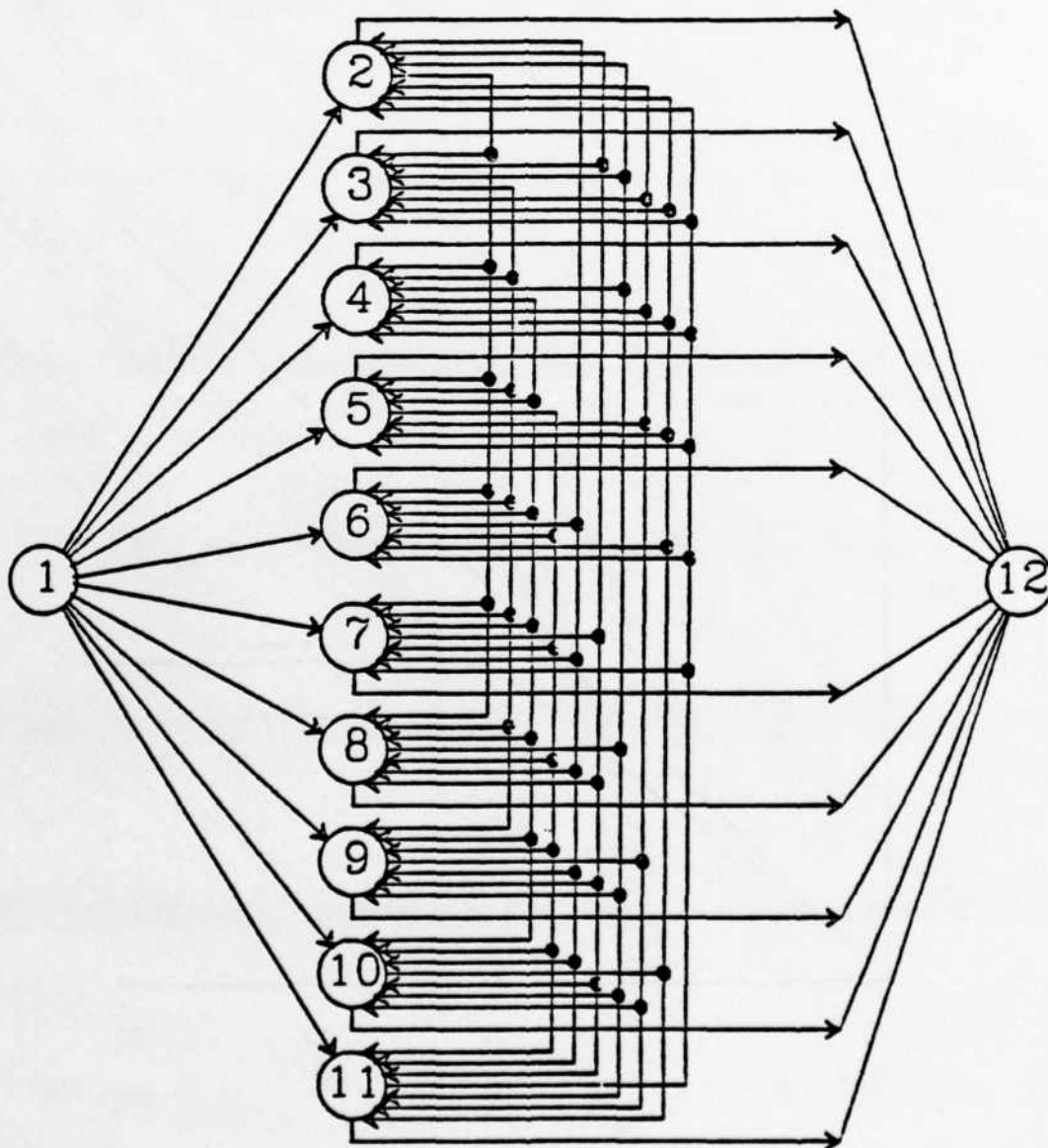


Figure 6-1 Program Structure of a SSIMD Filter with 10 PE's

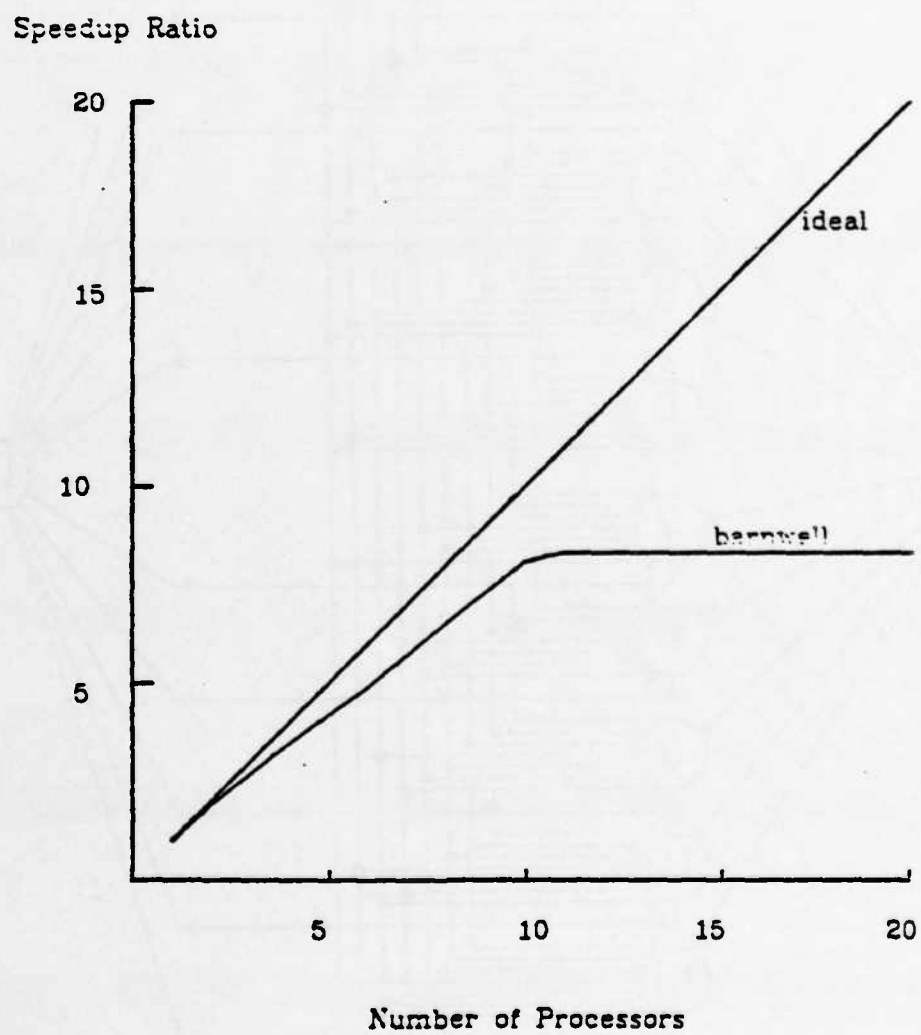


Figure 6-2 Speedup Ratio of SSIMD Structure

of increase in speed slows down and saturates after 11. Therefore, 10 PE's are PE optimum and 11 is speed optimum as mentioned in chapter 4. This figure verifies the summarized results stated in section 4-1.

Figure 6-3 shows the effect of transmission delay on the speed performance. The results are obtained by using 10 PE's whose structure is shown in Figure 6-1. The speed with null transmission delay is used as the reference and set to unity. A constant delay for each message transmission is simulated for the range from  $1 \mu s$  to  $50 \mu s$ . The speedup drops below 0.2 when the delay is  $50 \mu s$ . Better simulation results can be found in R. Fujimoto's dissertation[43] considering various type of interconnections as well as a finite bandwidth assigned to each communication link. The effect of multi-cast capability of each transmission link is also investigated.

### 6.1.2. Block State Filters

The filter described by equation (6-1) is simulated in the block state form with various block sizes. Figure 6-4 shows the program structure of the filter with block size  $L=6$ . Lines and arrows represent the actual data flow in the program. The floating lines and arrows are fed into a dummy data sink, which does not have any function and does not take any time. These floating lines and the dummy data sink are necessary so as to make all the cells similar, and hence only a single routine is required for the whole simulation. In the actual implementation, they may not be necessary. Task 1 reads input samples from an input file and then distributes them into several other tasks. It also sends '0' to all the tasks which need it. Task 29 takes final data from the output of matrix C and then writes into an output file.

Tasks 2 to 10 perform the matrix multiplication of matrix B. Each task performs a two by two matrix multiplication. The y output travels horizontally to matrix A and the x input travels downward to the data sink. Tasks 11, 12 and 13

## Barnwell Structure with 10 Processors

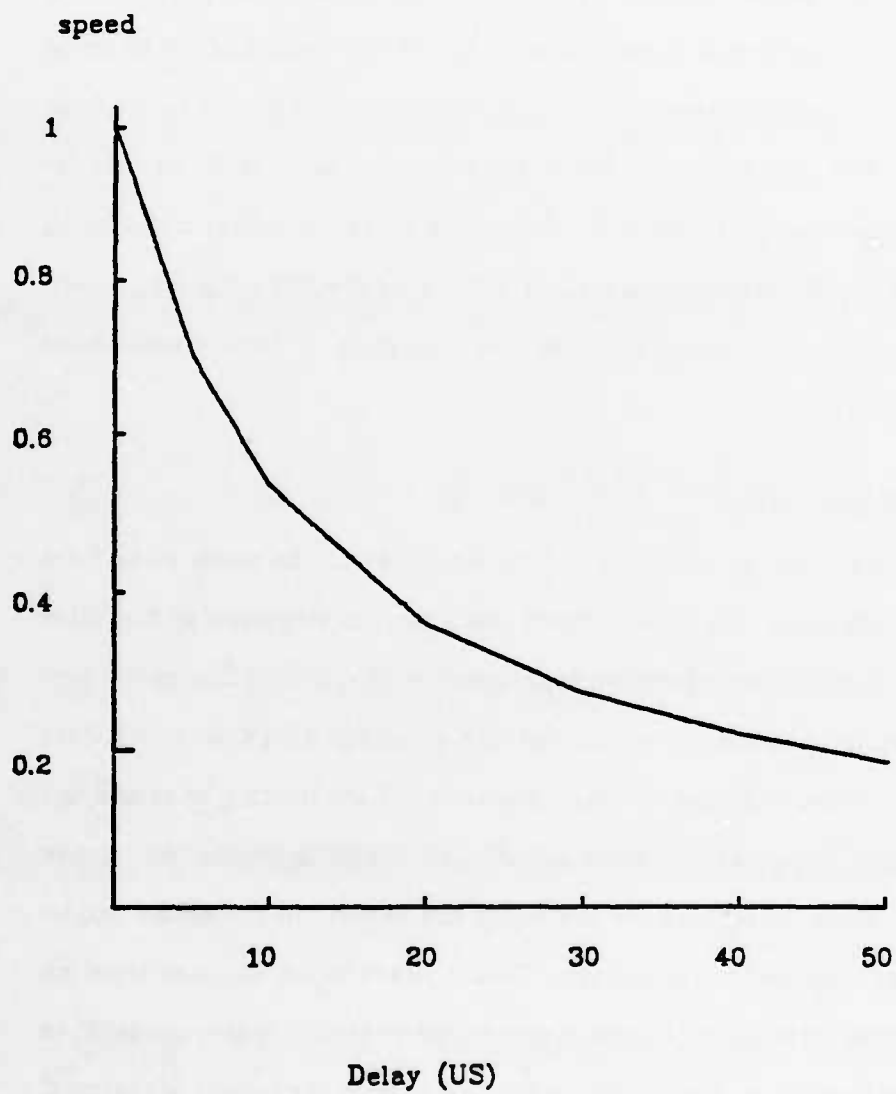


Figure 6-3 Transmission Delay Effect of SSIMD Structures

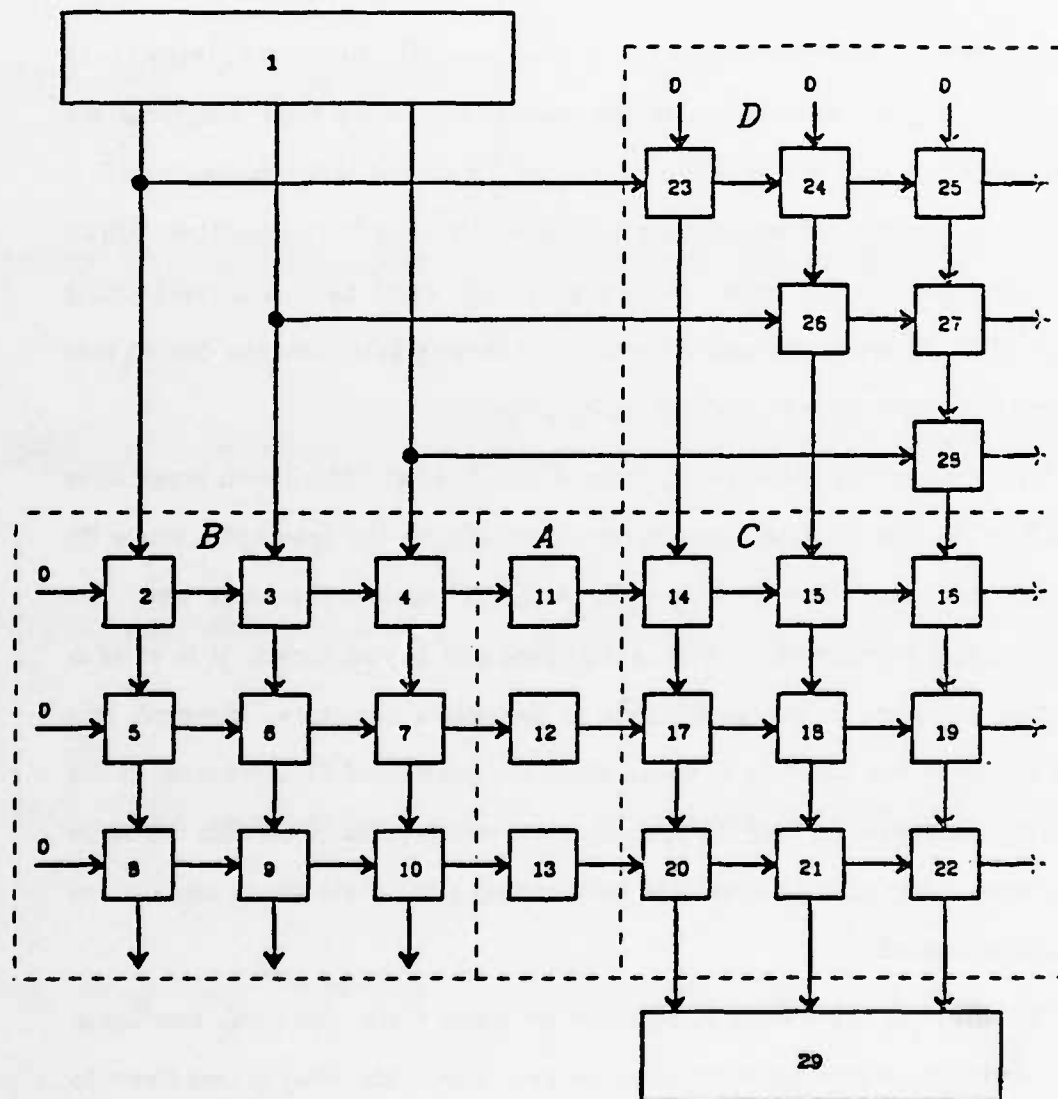


Figure 6-4 Program Structure of a Block State Filter with  $L=6$



perform the recursive operation and then add the results to the inputs from B. Their outputs are sent to matrix C as the input vector. The initial y inputs come from the output of matrix D and the output samples travel vertically. Since matrix D is lower diagonal, only six tasks are required instead of nine. Furthermore, the tasks on the diagonal, which are 23, 26 and 29, contain only three non-zero coefficients.

Only two routines are written for all the tasks. The function in tasks 11, 12 and 13 are slightly different from the others, and hence their programs are separate. Since only one program is written for all the tasks in matrices B, C and D, they should have exactly the same operation and I/O connection. SIMON does not allow floating links. Each export FIFO must have a corresponding import FIFO. Therefore, a sink is required to receive data from the floated output ports in order to make SIMON working properly.

Figure 6-5 shows the speedup ratio of this filter structure with block sizes from 2 to 10. The relative speed is measured against the speed of a single PE Barnwell structure. The speedup ratio is plotted against the block size. The corresponding number of PE from each block size is also shown. It is obvious that this structure is not so efficient as Barnwell's structure. However, this structure does not saturate in speed when the number of PE increases. If the block size increases further the speedup ratio curve would go up with the same slope without any limit. Hence, it is verified that block state filters can achieve any desired speed.

The effect of the transmission delay for block state filters was also simulated for the case of a constant delay on each link. This delay is less likely to happen than in Barnwell structure, because of the local interconnect. The simulation is done with block size 10 and the speed with null transmission is used as a reference. The overall speed drops slightly when the delay increases to 50  $\mu$ s

Speedup Ratio

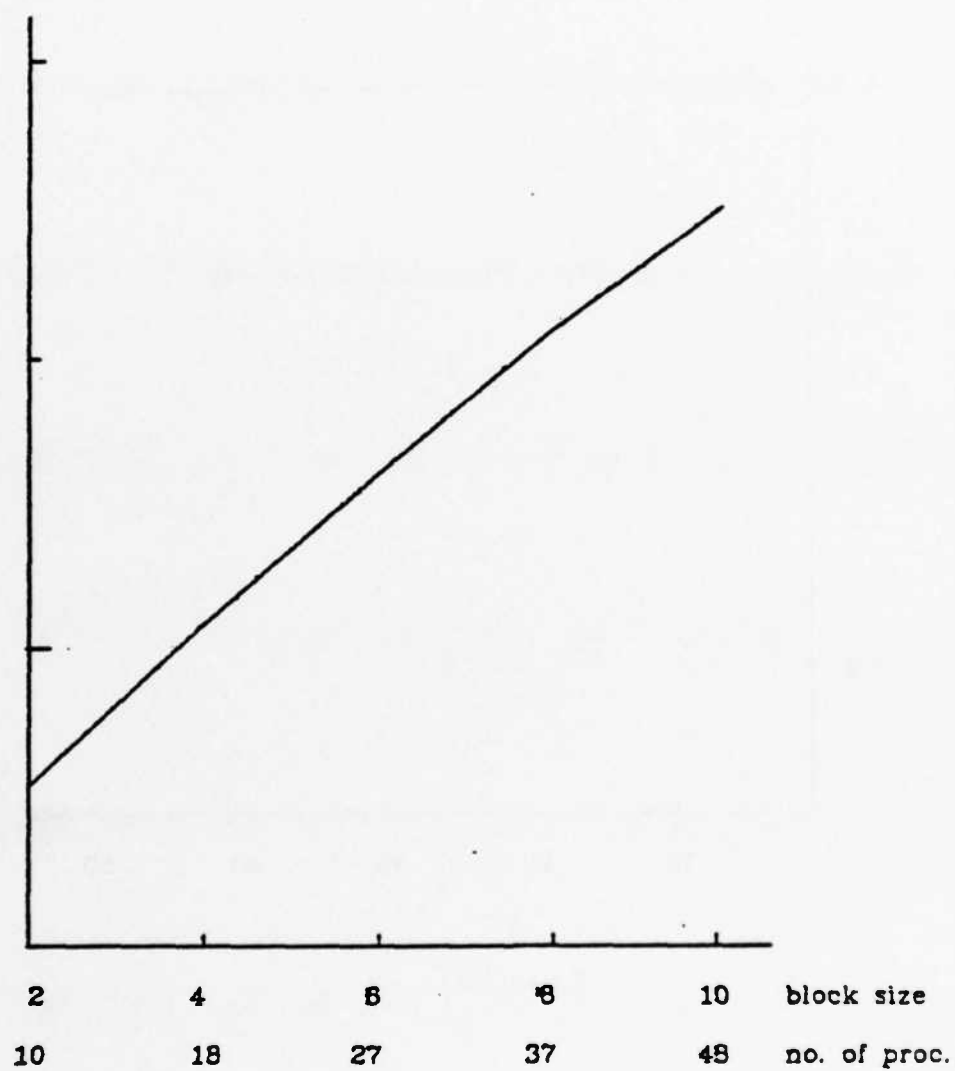


Figure 6-5 Speedup Ratio of a Block State Filter

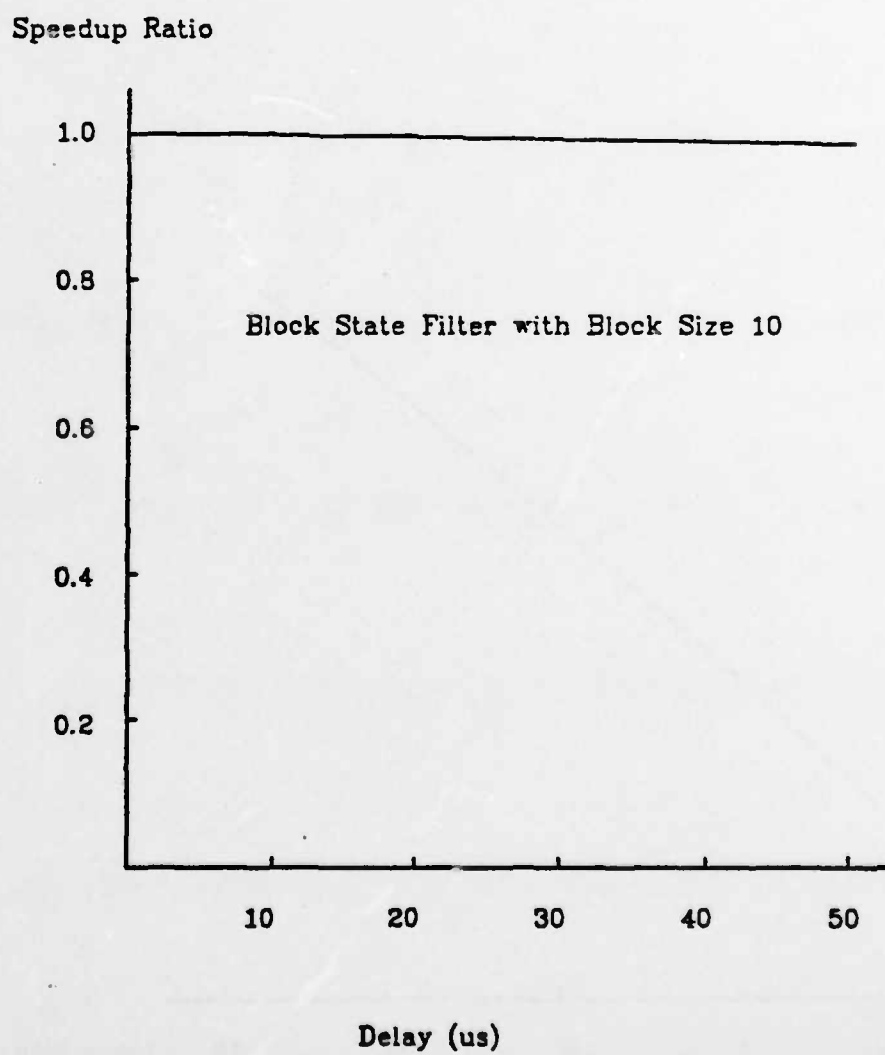


Figure 8-6 Transmission Delay Effect of a Block State Filter

(See Figure 6-6). This speed drop is due to the fact that the program for the recursive operations is different from the program for the rest of the cells. Hence, the execution time is not a constant for all the cells. Since SIMON is an event driven simulator, and there is no global clock to drive the function of each cell, the uneven execution time might come into effect when the delay is large. In the actual implementation, if a global clock is used, all the cells have the same execution time, and the overall speed will not drop with high transmission delays.

## 6.2. Roundoff Noise Simulation

Due to its random behavior, roundoff noise is not easy to simulate on a digital computer directly. However, with a little approximation, fairly reasonable results can be obtained. We will describe in this section our approach to the simulation and compare the results with the theoretical ones obtainable from equation (5-9).

Excited by a perfect sine wave, a filter should generate, at its output, a sine wave of the same frequency plus some roundoff noise. Observing this output waveform for a long enough time, we should be able to measure the output noise power by looking at the output spectrum. It is simply the overall output power minus the power of the output sine wave.

Before applying the above technique, however, two problems have to be resolved. First, a perfect sine wave is not obtainable on a digital computer. The input signal has to be quantized to fit on finite length registers or memories. Therefore, the output noise contains filter roundoff noise as well as the input quantization noise passing through the filter. Since the input precision is usually close to that of internal samples in most applications, these two noise components are too close to be distinguished from each other simply by looking at the output spectrum. One way to remedy this problem is to assign more bits to

the input samples than to the internal and output data. This can give us a nearly perfect sine wave and hence the input error has very little effect on the output noise.

Second, in order to have good results, the output noise has to be stationary. For the block state filter, the output noise is periodic stationary with the period of the block size. Hence, it is more reasonable to separately measure the noise power for each component in a block. The noise power of the  $i^{\text{th}}$  sample is measured by observing the spectrum of the sequence formed by the  $i^{\text{th}}$  samples of all the blocks. In order to ensure that the decimated output signal is a sinusoid, a higher sampling rate is required for the input signal. Furthermore, the number of different input values has to be large enough that the filter experiences rounding at all different signal levels. This has to be true so as to satisfy the assumption that the noise is uncorrelated from sample to sample.

With all the above restrictions in mind, we will introduce some mathematical basis to perform the simulation and then show the actual structure of the simulation. Finally, the simulation results of the test filter for various block sizes will be shown. The filter coefficients are obtained by grouping second order normal digital filters described in section 5.5.2.

### 6.2.1. Mathematical Formulation

Duttweiler and Messerschmitt[44] have shown a method to test A/D and D/A converters with digitally generated sinusoids. Their algorithm is also suitable for the roundoff noise measurement.

#### 6.2.1.1. Structure

Suppose the input sequence is represented as

$$x_k = A \sin(2\pi f_d kT) \quad (6-2)$$

where  $f_d$  is the frequency of this sine wave and  $T$  is the sampling period. The

output sequence  $\{y_k\}$  is also a sampled sine wave with frequency  $f_d$  but with different amplitude and phase plus a noise term

$$y_k = B \sin(2\pi f_d kT + \psi) + n_k \quad (6-3)$$

Suppose  $N$  samples of a sine wave are observed. These samples must contain an integer number of cycles in order to have a single line spectrum at the desired frequency. This is equivalent to

$$f_d NT = M$$

for some integer  $M$ . Hence, the available frequencies are of the form

$$f_d = \frac{M}{NT} \quad (6-4)$$

Substituting (6-4) into (6-2) gives

$$x_k = A \sin(2\pi Mk / N)$$

Furthermore, in order to have as many different values as possible, it is desired that there is only one cycle in these  $N$  samples. This is equivalent to constraining the two integers  $N$  and  $M$  to be relatively prime. Another constraint is that,  $f_d$  has to be less than  $\frac{1}{2T}$  so as to satisfy the nyquist sampling criterion. From equation (6-4),

$$\frac{M}{N} = f_d T \leq \frac{1}{2T} T = \frac{1}{2}$$

or

$$M \leq \frac{N}{2}$$

### 6.2.1.2. Noise Power Measurement

Taking the DFT of the output sequence  $y_k$  gives

$$Y_n = \sum_{k=0}^{N-1} y_k e^{-j2\pi nk/N}$$

The total power at all frequencies except the power at the sine frequency and dc is naturally called noise power (NP). Hence,

$$\begin{aligned}
 NP &= \sum_{n=1}^{N-1} |Y_n|^2 \\
 &= \sum_{n=0}^{N-1} |Y_n|^2 - |Y(0)|^2 - 2|Y(M)|^2
 \end{aligned} \tag{6-5}$$

According to the Parseval's theorem,

$$\sum_{n=0}^{N-1} y_n^2 = \frac{1}{N} \sum_{k=0}^{N-1} |Y_k|^2$$

Replace this equation into (6-5), we get

$$NP = \sum_{n=0}^{N-1} y_n^2 - |Y(0)|^2 - 2|Y(M)|^2 \tag{6-6}$$

For a block state filter with block size L, we have to modify equation (6-a) as

$$x_k = A \sin(2\pi Mk / NL)$$

where M and NL should be relatively prime. For the  $l^{th}$  term in the block, the output sequence looks like

$$\begin{aligned}
 y_k &= B \sin\left(\frac{2\pi M(kL+l)}{NL} + \psi\right) + n_{kL+l} \\
 &= B \sin\left(\frac{2\pi Mk}{N} + \frac{2\pi Ml}{NL} + \psi\right) + n_{kL+l} \\
 &= B \sin\left(\frac{2\pi Mk}{N} + \psi_l\right) + n_{kL+l}
 \end{aligned} \tag{6-7}$$

where  $\psi_l = \frac{2\pi Ml}{NL} + \psi$ . The phase angle  $\psi_l$  is constant for the  $l^{th}$  term, and hence will not contribute to the output power measurement. Therefore, equation (6-6) can still be used to calculate the  $l^{th}$  noise power in a block but where the frequency is deemed to be  $\frac{M}{NT}$  rather than  $\frac{M}{NLT}$ .

### 6.2.2. Simulation Routines

The high precision of the input samples makes the simulation a little complicated since it may happen that no existing fixed point variable can hold the even higher precision of the products of the input samples and the stored coefficients. Hence, a special variable has to be created in order to perform the high precision multiplication. Fortunately, the "C" program provides an excellent tool, which is called *struct* to solve this problem. Then, programs to



perform the high precision multiplication and addition have to be written. Finally, a rounding routine is required. When these routines are done, the noise simulation is straightforward in conjunction with the filter and the FFT routines.

#### 8.2.2.1. High Precision Multiplier and Adder

To generate a near perfect input sine wave, more bits must be assigned to the input samples than the filter coefficients and output samples. In the simulation program, 32 bits are assigned to the input samples and 16 bits to the coefficients and output data. The problem arising from this high precision is that the fixed point multiplier requires 47 bits to hold the precision of its output. This much higher precision is impossible to simulate on a VAX machine directly. Therefore, subroutines are written to emulate the long number multiplication.

The output from the multiplier is stored in an array of three 16-bit short integers which are grouped into a "struct" called `long_nu`.

```
typedef struct {
    short t[3];
} long_nu;
```

Due to the precision limitation on VAX machines, the input samples cannot be multiplied by the filter coefficients directly. Therefore, the input samples are divided into two 16-bit short integers and then multiplied by the coefficient. The two products are summed up after shifting.

Suppose the input sample  $x$  is divided into  $x_1$  and  $x_2$ , where  $x_1$  contains the leftmost 16 bits and  $x_2$  contains the rest. It is obvious that the most significant bit of  $x_2$  is no longer a sign bit. Let us store these two 16-bit numbers in two 32-bit integers  $x_1'$  and  $x_2'$  respectively. The left 16 bits of  $x_2'$  are all 0's and the left 16 bits of  $x_1'$  are the same as the Most Significant Bit (MSB) of  $x_1$  (This is called sign extension). Then

$$x = 2^{16} x_1' + x_2'$$

Suppose the coefficient is "a" and the product is  $y = ax$ . Since a is also a short

integer, we can obtain  $y$  as

$$y = 2^{16} ax_1 + ax_2$$

This is equivalent to shifting the product  $ax_1$  16 bits to the left and then adding it to  $ax_2$ .

As for the addition of two numbers, care must be taken when adding two products of different lengths. The product of the recursive state variables and the coefficients has only 31 bits instead of 47 bits. When adding up a 47 bit number to a 31 bit number, the most significant bit must be lined up rather than the least significant bit. This can be done by storing the 31 bit integer into the left two short integers,  $t[1]$  and  $t[2]$ , of `long_nu` and setting  $t[0]$  to 0. Similar to multiplication, the sign bits of the right two short integers are not sign bits any more. When added, these two bits must be treated as regular bits rather than sign bits.

#### 6.2.2.2. Rounding Routine

Rounding can be done by a simple truncation after adding a bit "1" to a proper position. If the leftmost 16 bits are to be preserved after rounding, the bit "1" should be added to the 17<sup>th</sup> bit from the left. It is always added to the bit next to the Least Significant Bit (LSB) of the rounded number. Another thing that has to be taken care of is the positioning of the decimal point. Since the coefficients might range from a very small number to a very large one, the designer might have to allocate the decimal point so as not to overflow the largest number. When performing the rounding, this decimal point also has to be considered in order to obtain the correct result, otherwise the state variables in the next cycle will not be valid any longer.

#### 6.2.3. Simulation Results

Table 6-1 shows the output roundoff noise power of the filter characterized

by equation (6-1) with various block sizes. the noise powers are obtained by calculating equation (5-9). All the coefficients have to be truncated to fit on 16 bit registers before plugging into (5-9). On the other hand, in order to maintain high precision during calculation, the truncated numbers are expressed by 64 floating point numbers. Table 6-2 shows the roundoff noise by the simulation technique described in the previous sections. It is easily seen that the simulation results are close to those from the calculation.

The simulation results also verify that the noise performance improves as block size increases. It is also apparent that the noise power of the first sample is larger than any other component in the block. The simulated roundoff noise of a cascade form filter is shown at the end of Table 6-2. It is clear that this noise power is close to that of the state filter with block size 1. However, it is impossible to modify this SISO form to improve the roundoff noise performance. Hence, block state filters are better structures considering the roundoff noise behavior.

Roundoff Noise Calculation							
Block Size	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7
1	0.6918						
2	0.5429	0.2524					
3	0.5073	0.2163	0.1349				
4	0.4950	0.2099	0.1311	0.1058			
5	0.4897	0.2068	0.1291	0.1045	0.0951		
6	0.4870	0.2051	0.1280	0.1037	0.0945	0.0903	
7	0.4854	0.2041	0.1272	0.1032	0.0942	0.0901	0.0877

Table 6-1 Roundoff noise of block state filters obtained from calculation

### 6.3. Conclusions

The speed performance of Barnwell filter as well as block state filter was simulated for various numbers of PE's and block sizes. The results verified the analyses in Chapter 4 for both structures. The speed of Barnwell filter saturates

Roundoff Noise Simulation							
Block Size	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7
1	0.7022						
2	0.5461	0.2257					
3	0.5201	0.2165	0.1331				
4	0.5023	0.2143	0.1293	0.1116			
5	0.4975	0.2099	0.1217	0.1030	0.0917		
6	0.4906	0.1963	0.1232	0.1008	0.0955	0.0904	
7	0.4923	0.2220	0.1319	0.1090	0.0932	0.0939	0.0877

Cascade Form: 0.6979

Table 6-2 Roundoff noise of block state filters obtained from simulation

if the number of PE's used is beyond some value, which is 11 in our specific simulation example. When the transmission delay comes into effect, the speedup ratio of this structure drops drastically with delay time. This is exactly what we expected in Chapter 4. As for the block state filter case, the speedup ratio does not saturate with large block size, or equivalently more PE's. Furthermore, the transmission delay has very little effect on the overall speed. The unlimited speedup ratio and the insensitivity to the transmission delay make this structure far better than the Barnwell structure in VLSI application.

The roundoff noise simulation of block state filters also verified the derivation in the previous chapter. the improvement in this noise performance with increased block size can be easily seen from the simulation results. The scalar state filter structure was also verified to be equivalent to the cascade form structure. As the block size increases, the block state structure will outperform the cascade form.

## CHAPTER 7

## CONCLUSIONS

Several algorithms and structures for implementing digital filters with a high degree of parallelism have been presented. The motivation behind the use of a parallel or pipelining technique is to increase the sampling rate is the desire to implement the filter in VLSI circuits. VLSI can provide much more computational hardware in a single chip, but without commensurate increases in speed. Parallel algorithms can effectively increase the throughput rate by utilizing the high density of the VLSI circuits. Actually, the effectiveness in the speed performance improvement is beyond what we expected, since the sampling rate can be increased indefinitely by adding hardware. The sampling rate is thus limited only by the die area, and not the speed of the hardware.

The block processing of the input samples is proposed for the realization of both FIR and IIR filters. Block processing has been shown to be effective in increasing the internal parallelism of a given filter. Together with the systolic idea, block processing can achieve very high speed with only local interconnection among PE's. This is straightforward for FIR filter realization. For IIR filters, state equations are employed to model the filtering function. With state equations, the feedback computation has a fixed rate for a given filter and also can be decomposed into smaller chunks, which are of fixed size and are independent of the filter order. The feedforward operation, on the other hand, can be realized by two dimensional arrays. Block state filters break the limitation on the highest speed imposed by the recursive operation. Higher speed can be achieved by a larger block size and a simple expansion in the overall structure without complicating the communication environment.

IIR filters are known to require much less computation than FIR structures to achieve a given frequency response. If linear phase is not essential, IIR filters are preferred in order to save hardware. However, as block size in an IIR filter increases to process signals with higher sampling rate, the average computation per output sample also increases. On the other hand, the average computation stays the same as the block size increases for FIR filter structures. Since the average computation per output sample is a good indicator of the hardware size required, FIR filters should be used for very high speed filters. It is shown in Chapter 3 that FIR filters can also be realized with only local interconnections.

Another factor that will affect the filter performance as well as the hardware size is the finite word length effect. The block state filter is shown to have very low roundoff noise. Further, the roundoff noise performance improves as the block size increases. Hence, although the average computation increases, the roundoff noise is lower for higher speed processing. Another effect is that the poles move toward the origin as the block size increases. Therefore, the filter is less likely to become unstable when quantizing the filter coefficients.

In chapter 3, the applications of the two dimensional systolic array and the block state filters to the computer graphics and decimation as well as interpolation are illustrated. However, the problem of the coefficient update for various rotation, scaling and translation of computer graphs is left open. Further research is required before the systolic approach can be applied to the computer graphics area. Furthermore, the two applications described by no means exhaust all the possibilities for their application. Further research is necessary to find more applications.

Due to advances in IC technology, adaptive algorithms become more economical for sophisticated signal processing systems. Therefore, more adap-

tive or time-varying filters are used to achieve higher performance or to reduce the bit rate. These filters, however, are difficult for parallel processing, since the filter coefficients are changing with time. Furthermore, the output samples are usually fed back to adapt the filter coefficients and this adaptation has to be completed in a fixed time interval. Further effort in applying parallelism to adaptive signal processing is therefore needed.



## References

1. Alan V. Oppenheim and Ronald W. Schaffer, in *Digital Signal Processing*, p. 218, Prentice-Hall, 1975.
2. Richard M. Fujimoto, *A User's Manual for a Multiprocessor Simulator*, UC Berkeley, Feb. 11, 1982.
3. Wang Ho Yu, "LU Decomposition on a Multiprocessing System with Communications Delay," *Ph.D. Dissertation*, UC Berkeley, In Preparation.
4. David G. Messerschmitt, *Blosim - a Block Simulator*, UC Berkeley, Jun. 1982.
5. Leland B. Jackson, "Round-off-noise Analysis for Fixed Point Digital Filters Realized in Cascade or Parallel Form," *IEEE Trans. Audio Electroacoust.*, vol. AU-18, pp. 107-122, Jun. 1970.
6. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, ed. I. S. Duff and G. W. Stewart, pp. 256-282, Society for Industrial and Applied Mathematics, 1979.
7. H. T. Kung, "Special-Purpose Devices for Signal and Image Processing: an Opportunity in Very Large Scale Integration (VLSI)," in *Proceedings of the Society of Photo-Optical Instrumentation Engineers*, ed. Tien F. Tao, Real-Time Signal Processing III, vol. 241, pp. 76-84, July 1980.
8. H. T. Kung, "Why Systolic Architectures?," *IEEE Computer*, vol. 15, no. 1, pp. 37-43, Jan. 1982.
9. T. P. Barnwell, III, *Optimal Implementations of Recursive Signal Flow Graphs on Synchronous Multiprocessor Architectures in SSIMD Mode*, Paper in preparation.
10. T. G. Stockham, Jr., "Highspeed Convolution and Correlation," 1966 Spring Joint Computer Conf., AFIPS Proc., vol. 28, pp. 229-233, Washington, D.

C.:Thompson, 1966.

11. B. Gold and K. L. Jordan, "A Note on Digital Filter Synthesis," *Proceedings IEEE (Lett.)*, vol. 65, pp. 1717-1718, Oct. 1968.
12. H. B. Voelcker and E. E. Hartquist, "Digital Filtering Via Block Recursion," *IEEE Trans. Audio and Electroacoustics*, vol. AU-18, pp. 169-176, June 1970.
13. Charles S. Burrus, "Block Implementation of Digital Filters," *IEEE Trans. Circuit Theory*, vol. CT-18, pp. 697-701, Nov. 1971.
14. R. Gnanasekaran and Sanjit K. Mitra, "A Note on Block Implementation of IIR Digital Filters," *Proceedings IEEE (Lett.)*, vol. 65, pp. 1063-1064, July 1977.
15. Sanjit K. Mitra and R. Gnanasekaran, "Block Implementation of Recursive Digital Filters --New Structures and Properties," *IEEE Trans. Circuits and Systems*, vol. CAS-25, pp. 200-207, April 1978.
16. Charles A. Desoer, in *EECS 222 Lecture Notes*, Department of Electrical Engineering and Computer Sciences, U. C. Berkeley.
17. Leland B. Jackson, Allen G. Lindgren, and Young Kim, "Optimal Synthesis of Second-Order State-Space Structures for Digital Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-26, pp. 149-153, Mar. 1979.
18. C. W. Barnes and A. T. Fam, "Minimum Norm Recursive Digital Filters That Are Free of Overflow Limit Cycles," *IEEE Trans. Circuits and Systems*, vol. CAS-24, pp. 569-574, Oct. 1977.
19. Adly T. Fam and Casper W. Barnes, "Nonminimal Realizations of Fixed-Point Digital Filters That Are Free of All Finite Word-Length Limit Cycles," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. ASSP-27, pp. 149-153, Apr. 1979.
20. Casper W. Barnes, "Roundoff Noise and Overflow in Normal Digital Filters,"

- IEEE Trans. Circuits and Systems*, vol. CAS-26, pp. 154-159, Mar. 1979.
21. C. W. Barnes and T. Miyawaki, "Roundoff Noise Invariants in Normal Digital Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-29, pp. 251-256, Apr. 1982.
  22. Casper W. Barnes and S. Shinnaka, "Block Shift Invariance and Block Implementation of Discrete-Time Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-27, pp. 667-672, Aug. 1980.
  23. Jan Zeman and Allen G. Lindgren, "Fast Digital Filters with Low Round-Off Noise," *IEEE Trans. Circuit and Systems*, vol. CAS-28, pp. 716-723, Jul. 1981.
  24. S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Trans. on Computers*, vol. C-31, pp. 1054-1066, Nov. 1982.
  25. J. D. Foley and A. V. Dam, in *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.
  26. Robert A. Meyer and Charles S. Burrus, "A Unified Analysis of Multirate and Periodically Time-Varying Digital Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-22, pp. 162-168, Mar. 1975.
  27. Robert A. Meyer and Charles S. Burrus, "Design and Implementation of Multirate Digital Filters," *IEEE Trans. ASSP*, vol. ASSP-24, pp. 53-58, Feb. 1976.
  28. R. W. Schafer and L. R. Rabiner, "A Digital Processing Approach to Interpolation," *Proceeding IEEE*, vol. 81, pp. 692-702, Jun. 1973.
  29. R. E. Crochiere and L. R. Rabiner, "Optimum FIR Digital Filter Implementations for Decimation, Interpolation, and Narrow-Band Filtering," *IEEE Trans. on ASSP*, vol. ASSP-23, pp. 444-456, Oct. 1975.
  30. R. E. Crochiere and L. R. Rabiner, "Further Considerations in the Design of

- Decimations and Interpolators," *IEEE Tran. on ASSP*, vol. ASSP-24, pp. 296-311, Aug. 1976.
31. Jan Zeman and Allen G. Lindgren, "Fast State-Space Decimator with Very Low Round-off Noise," *Signal Processing*, vol. 3, pp. 377-388, Oct. 1981.
  32. M. G. Bellanger, J. L. Daguet, and G. P. Lepagnol, "Interpolation, Extrapolation, and Reduction of Computation Speed in Digital Filters," *IEEE Trans. ASSP*, vol. ASSP-22, pp. 231-235, Aug. 1974.
  33. Charles S. Burrus, "Block Realization of Digital Filters," *IEEE Trans. Audio and Electroacoustics*, vol. AU-20, pp. 230-235, Oct. 1972.
  34. R. E. Crochiere and A. V. Oppenheim, "Analysis of Linear Digital Networks," *Proceedings IEEE*, vol. 63, pp. 581-594, Apr. 1975.
  35. L. R. Rabiner, J. F. Kaiser, O. Herrmann, and M. T. Dolan, "Some Comparisons Between FIR and IIR Digital Filters," *Bell Syst. Tech. Journal*, vol. 53, pp. 305-331, Feb. 1974.
  36. M. Renfors and Y. Neuvo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints," *IEEE Trans. Circuit and Systems*, vol. CAS-28, pp. 198-202, Mar. 1981.
  37. Leland B. Jackson, "On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 159-183, Feb. 1970.
  38. Casper W. Barnes and Shinji Shinnaka, "Finite Word Effects in Block-State Realizations of Fixed-Point Digital Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-27, pp. 345-349, May 1980.
  39. Syeng Y. Hwang, "Roundoff Noise in State-Space Digital Filtering: A General Analysis," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-24, pp. 256-262, June 1976.

40. Clifford T. Mullis and Richard A. Roberts, "Synthesis of Minimum Roundoff Noise Fixed Point Digital Filters," *IEEE Trans. Circuits and Systems*, vol. CAS-23, pp. 551-562, Sep. 1976.
41. Syeng Y. Hwang, "Minimum Uncorrelated Unit Noise in State-Space Digital Filtering," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. ASSP-25, pp. 273-281, Aug. 1977.
42. C. M. Rader and B. Gold, "Effects of Parameter Quantization on The Poles of a Digital Filter," *IEEE Proceedings*, vol. 55, pp. 888-889, May 1967.
43. Richard M. Fujimoto, "VLSI Communications Components for Multicomputer Networks," *Ph.D. Dissertation*, UC Berkeley, Aug. 1983.
44. D. L. Duttweiler and D. G. Messerschmitt, "Analysis of Digitally Generated Sinusoids with Application to A/D and D/A Converter Testing," *IEEE Trans. Communications*, vol. COM-26, pp. 669-675, May 1978.

# VLSI COMMUNICATION COMPONENTS FOR MULTICOMPUTER NETWORKS

*Richard Masao Fujimoto*

## ABSTRACT

Advances in microprocessor technology will soon make general purpose computing systems composed of thousands of VLSI processors economically feasible. A high-performance communication system to interconnect these processors is of crucial importance to exploit the parallelism inherent in applications such as circuit simulation and signal processing. This thesis discusses issues in the design of universal VLSI communication components to be used as the building blocks for constructing robust, high-bandwidth, point-to-point networks. The components provide enough flexibility to serve a wide variety of multicomputer configurations and applications. They feature special purpose hardware to implement communication functions traditionally implemented with network software.

A communication network constructed from the proposed components is modeled as a set of nodes (components) connected by bidirectional communication links. Because of technological constraints, the total I/O bandwidth of each node is limited to some fixed value, and assumed to be equally divided among the attached links. Increasing the number of links per component leads to a reduction in the average number of hops between nodes, but at the cost of reduced link bandwidth. This "hop count / link bandwidth" tradeoff is examined in great detail through M/M/1 queueing models and simulations using traffic loads generated by parallel application programs. These results indicate that a small number of links should be used. It is also found that a significant improvement in performance is obtained if a component is allowed to *immediately* begin forwarding a message when the selected output link becomes idle, regardless of

whether or not the end of the message has arrived. Finally, mechanisms which efficiently transmit a single message to multiple destinations are seen to have a significant impact on performance in programs relying on global information.

The complexity of the circuitry required to implement a communication component is examined. Schemes for providing hardware support for communication functions — routing, buffer management, and flow control — are presented. Estimates of the number of buffers and the degree of multiplexing on each communication link are determined. The amount of circuitry to implement a communication component is computed, and it is seen that the proposed communication component could be implemented with technology available today. Design recommendations for the implementation of such a component are made.

Richard M. Fujimoto 8/25/83

Richard M. Fujimoto

Carlo H. Séquin 8/25/83

Carlo H. Séquin  
(committee chairman)



## TABLE OF CONTENTS

<b>CHAPTER ONE - INTRODUCTION .....</b>	<b>1</b>
1.1. The Concept: Modular, High-Bandwidth Communication Networks .....	3
1.2. Definition of Terms .....	6
1.3. Previous Work in Communication Networks .....	8
1.3.1. Loosely Coupled Computer Networks .....	8
1.3.2. Interconnection Networks for Closely Coupled Multiprocessors .....	10
1.4. Overview of Thesis .....	17
<b>CHAPTER TWO - PERFORMANCE EVALUATION STUDIES .....</b>	<b>20</b>
2.1. VLSI Constraints .....	20
2.1.1. Area .....	21
2.1.2. Power .....	21
2.1.3. Pins .....	23
2.1.4. Summary of Constraints .....	24
2.2. Analytical Studies .....	24
2.2.1. Assumptions .....	26
2.2.2. Model I: Cluster Nodes .....	28
2.2.2.1. Queueing Model .....	30
2.2.2.2. Delay .....	35
2.2.2.3. Bandwidth .....	36
2.2.3. Model II: Networks with a Fixed Number of Components .....	44

2.2.3.1. Queueing Model .....	45
2.2.3.2. Delay .....	48
2.2.3.3. Bandwidth .....	54
2.2.4. M/G/1 Queueing Models .....	55
2.2.5. Summary of Analytic Results .....	57
<b>CHAPTER THREE - SIMULATION STUDIES .....</b>	<b>59</b>
3.1. The Simulator: Simon .....	59
3.2. Assumptions .....	60
3.3. The Application Programs .....	64
3.3.1. Barnwell Filter Program (global SISO, 12 tasks) .....	67
3.3.2. Block I/O Filter Program (local SISO, 23 tasks) .....	69
3.3.3. Block State Filter Program (local SISO, 20 tasks) .....	70
3.3.4. FFT Program (local PIPO, 32 tasks) .....	72
3.3.5. LU Decomposition (global PIPO, 15 tasks) .....	74
3.3.6. Artificial Traffic Loads (global PIPO, 12 tasks) .....	75
3.4. Communication Delays .....	75
3.5. Issues Under Investigation .....	79
3.6. Simulation Results on Cluster Node Networks .....	81
3.6.1. Fully Connected Networks .....	82
3.6.2. Full-Ring Tree Networks .....	87
3.6.3. Butterfly Networks .....	93
3.6.4. Ring Networks .....	98
3.6.5. Conclusions for Cluster Node Networks .....	102
3.7. Simulation Results on Networks with a Fixed Number of Components .....	103

3.7.1. Lattice Topologies .....	104
3.7.2. Tree Topologies .....	113
3.7.3. De Bruijn Networks .....	113
3.7.4. Conclusions for Networks with a Fixed Number of Components .....	120
3.8. Influence of the Mapping of Tasks to Processors .....	120
3.9. Precision of the Simulations Results .....	135
3.10. Summary of Simulation Studies .....	138
<b>CHAPTER FOUR - DESIGN AND IMPLEMENTATION OF COMMUNICATION COMPONENTS .....</b>	<b>139</b>
4.1. Transport Mechanisms .....	139
4.2. A Virtual Circuit Based Communication System .....	144
4.2.1. Virtual Circuits .....	144
4.2.2. Virtual Channels .....	145
4.2.3. Routing Hardware .....	147
4.2.4. Packet Types and Formats .....	148
4.3. Key Functions of the Communication Component .....	149
4.3.1. Routing .....	150
4.3.2. Buffer Management .....	153
4.3.3. Flow Control .....	155
4.4. Implementation of VLSI Communication Components .....	159
4.4.1. Routing Hardware .....	159
4.4.2. A Y-Component Design .....	161
4.4.2.1. Buffer Management Hardware .....	164
4.4.2.2. Flow Control Hardware .....	165

4.4.3. Deficiencies in the Y-Component Design .....	165
4.4.4. An Alternative Design .....	167
4.4.4.1. Buffer Management Hardware .....	171
4.4.4.2. Flow Control Hardware: Send/Acknowledge Protocol .....	176
4.4.4.3. Flow Control Hardware: Remote Buffer Management .....	179
4.5. Evaluation of Communication Component Parameters .....	181
4.5.1. Number of Virtual Channels .....	182
4.5.2. Amount of Buffer Space .....	190
4.5.2.1. Buffer Space: Deadlock Considerations .....	191
4.5.2.2. Buffer Space: Performance Considerations .....	195
4.6. Complexity of the Communication Component .....	206
<b>CHAPTER FIVE - CONCLUSIONS .....</b>	<b>211</b>
<b>REFERENCES .....</b>	<b>213</b>

## CHAPTER ONE

### INTRODUCTION

The processing power of a general purpose computing system can be increased in two ways. One approach, which has the advantage that old software can be re-used, is to increase the speed of an existing computer system by technological means without altering the basic organization of hardware components. Much of this effort focuses on the development of very high speed electrical circuits through the use of new materials, e.g. Joseph junctions [Ghee82] or gallium arsenide [Long80]. The primary mode of operation in such a system is *sequential*, although limited amounts of parallelism may be employed in certain portions of the processor. The huge investment in existing software fuels the effort to make this approach commercially viable.

The second approach to building high-performance computer systems relies on a more general exploitation of parallelism, e.g. by using a large pool of relatively inexpensive computers that operate in *parallel* to solve a large problem which has been decomposed into a number of smaller subproblems. Advances in integrated circuit technology have made this approach feasible by allowing the construction of chips using a very large scale of integration (VLSI) to pack hundreds of thousands of transistors onto a small piece of silicon. It is in this latter approach that VLSI technology can have a truly dramatic impact in the structure of tomorrow's computing systems. This thesis will focus on the exploitation of parallelism to achieve high performance, and in particular, on the hardware necessary to support high bandwidth communications among thousands of processors.

A key design parameter of multicomputer systems (systems composed of more than one processor interconnected by a communication network) is processor *granularity*, i.e. the size and capability of the individual processing elements. At one end of the spectrum, each processing element is very small and limited in capability, allowing an entire multiprocessor system to be placed on a single chip. Examples are the special purpose systolic array processors which are particularly suitable for high-throughput signal processing applications [Kung80, Kung82], the 'tree-machine' developed at Caltech [Brow80], and the Boolean Vector Machine proposed by Wagner [Wagn83]. Since the unit to be replicated is small, often consisting of only an arithmetic unit and a few data registers, the granularity of the system is very fine. The other extreme, using very large granules, is exemplified by such supercomputers as the S1 which employs a few large, high-performance processors [Widd80]. Each processor consists of thousands of integrated circuit chips. Commercially available multiprocessor systems built by IBM [Ensl74a] or UNIVAC [Ensl74b] also belong to this category.

Earlier work in the X-tree project [Desp78, Sequ78] advocated an intermediate granule size equal to that of a single VLSI chip. For a *general purpose* system, some minimum complexity is required in each processing element to allow enough flexibility to enable several to cooperate productively across a wide range of applications. The simple processor advocated by the "small granule" approach is too small a building block for a general purpose computer. On the other hand, a very large granule size forces closely coupled components such as a processor and its associated memory to be implemented on separate chips, thus increasing the performance penalties resulting from off-chip communications. An intermediate granule size equivalent to a single-chip microprocessor and its memory forms an entity with enough processing power for general-

purpose computing, but is still small enough to be implemented on a single chip.

Advances in VLSI technology are making general-purpose computing systems composed of thousands of processors economically feasible. The processors, however, comprise only a portion of the system. The communication system that interconnects the processors is of equal importance. The performance of many multiprocessor systems has been limited by insufficient inter-processor input/output (I/O) bandwidth. Furthermore, the communication system may dominate the hardware cost. In Cm\*, for example, the hardware responsible for setting up the communication paths (i.e. the k-maps) was considerably more expensive than that used in the processors [Swan77b]. It is clearly desirable to also exploit VLSI technology to reduce the cost of the switching hardware. This thesis discusses issues in the design of universal VLSI switching components to be used as the building blocks for robust, high-bandwidth, communication networks with enough flexibility to serve a wide variety of multicomputer configurations and applications.

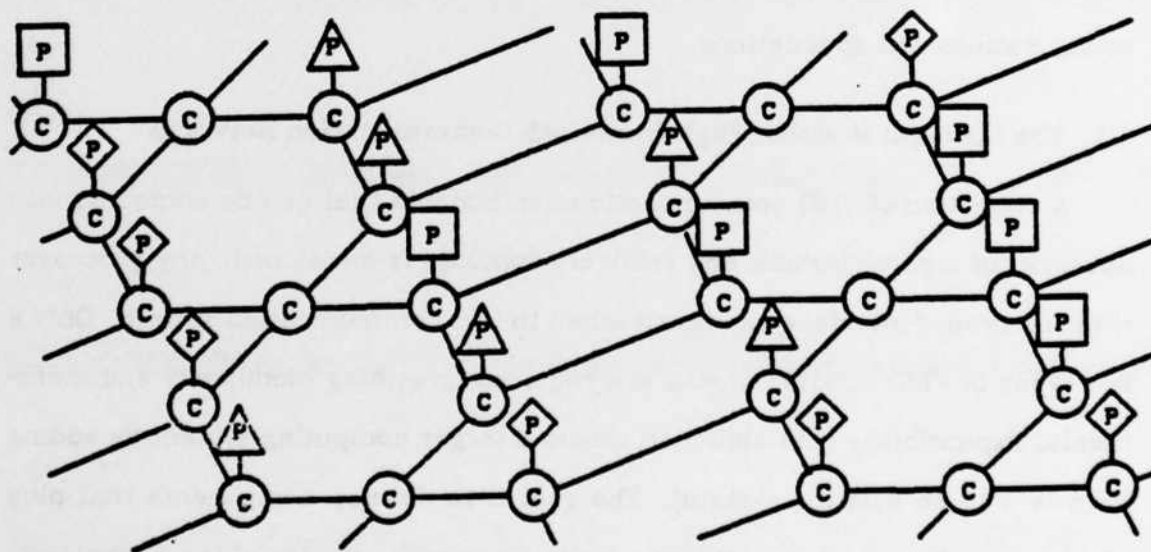
### **1.1. The Concept: Modular, High-Bandwidth Communication Networks**

A collection of VLSI communication components that can be combined into networks of high bandwidth and arbitrary topology is envisioned. Any processor with the proper interface can be attached to this communication system. Only a few types of VLSI building blocks are required, providing modularity and incremental expansibility (the ability to create a larger computing system by adding hardware to an existing system). The goal is to develop components that plug together easily and completely hide from the user the details of the information transfer within the network. Just as the telephone system hides from the user the details of routing calls and transferring voice information, these new communication modules handle the low-level details of transferring data by providing circuitry to perform communication functions such as handshaking,



message routing, buffering, and flow control. For the system designer, the lowest level primitive that must be dealt with is the information packet or the block of data to be transmitted. For the user of the final system, the network provides end-to-end communications much like the telephone system.

Figure 1.1 gives a conceptual view of such a system, divided into a communication domain (C) using these VLSI communication components, and a processor domain (P) dedicated primarily to the user's computations. Required properties of the communication domain include unrestricted network topology, modularity, incremental expansibility, decentralized control, and the ability to recover from certain classes of failures. Low-latency, high-bandwidth communications are required to achieve good performance in applications such as circuit simulation and signal processing. This research will focus on networks using



**Figure 1.1.** *Separation of a Multicomputer into a Communications Domain (C) and a Processor Domain (P).*

dedicated links. The proposed communication domain is designed to support high-performance communications among a large number, possibly thousands, of processors.

The proposed components perform several basic "store-and-forward" communication functions. Each component receives messages from any attached processor(s) and from other communication components. Before a message can be forwarded to the next component/processor, an output link must be selected via some *routing algorithm*. After an output link has been selected, the message is forwarded over this link. To handle conflicts which occur when more than one message is routed over the same output link at the same time, buffers are provided to hold waiting messages. Each communication component must provide circuitry for managing these buffers. Finally, to avoid losing messages when the buffer space in a component is exhausted, a *flow control* mechanism is required to throttle arriving traffic. Details of mechanisms which perform these functions are discussed in chapter 4, as well as estimates of the amount of circuitry required to implement them.

The types of processors used in the processor domain may vary depending on the application, but the interface to the communication system is standardized. This separation of the communication domain and the computation domain relieves the processors of much of the overhead associated with the forwarding of messages destined for different nodes. It makes possible the development of general purpose communication hardware that is suitable for a wide range of applications, and also provides the flexibility to construct heterogeneous systems containing many different types of specialized processors.

One may note the similarity between the components described here and communication processors used in loosely coupled computer networks. An example of such a processor is the Interface Message Processor (IMP) used in

the ARPANET, a computer network linking several major universities and institutions around the world [Hear70]. Indeed, many problems associated with loosely coupled computer networks (e.g. routing, buffering, flow control) also appear in this context. However, our design is not merely a scaled down version of the ARPANET. The key differences arise from the aim at higher bandwidth and lower latency, intrinsically lower error and failure rates within the communication hardware, and the envisioned implementation in VLSI.

## 1.2. Definition of Terms

A number of terms will be used throughout this thesis. In order to avoid confusion, their meaning in the context of this report will now be defined.

First, each *message* sent into the communication domain consists of some number of fixed length *packets*. Communication components deal exclusively with packets. Here, it is often the case that each message fits into a single packet, so the two terms will be used interchangeably when no confusion arises.

Each communication component contains some number of *ports* and *links*. A link refers to the collection of wires connecting a communication component to another such component or to a computation processor. The link is external to the chip. A port is the circuitry within the chip which drives data onto, and receives data from the link. When necessary, the distinction is made between an *input port*, which receives data entering the chip, and an *output port*, which sends data away from the chip. Each link is bidirectional and full duplex, i.e. each may simultaneously carry traffic in both directions. There is exactly one link attached to each port, so when referring to the "number of ports/links", the two terms are used interchangeably.

A *virtual circuit* refers to an end-to-end connection from one processor (say A) through a certain number of switching components to another processor

(B). Here, a virtual circuit (or circuit for short) refers to the directed path through the network from A to B. As will be discussed in chapter 4, virtual circuits must be "established" before messages can be sent, and all data sent on the same circuit follow the same path. In order to distinguish data on different virtual circuits which are using the same physical link, each link is divided into some number of *virtual channels*, with each channel carrying data for one circuit. Thus, a virtual circuit is a sequence of channels from one processor to another.

In the discussion presented above, virtual circuits were defined to have a single source and destination processor. An exception to this is a *multicast circuit* which has a single source, but more than one destination. A message sent on a multicast circuit is replicated within the network, and a separate copy is received by each destination processor. Such a mechanism is useful in applications requiring the same data to be distributed to several other processors, as will be discussed in chapter 3.

Another term used extensively in this thesis is *virtual cut-through* [Kerm79]. This refers to a mechanism in which the forwarding of data packets can begin as soon as the packet *header* (here, the first byte) arrives, if the proper outgoing link is idle. Without cut-through, forwarding would have to be delayed until the entire packet has arrived in its buffer. It will be seen that this immediate forwarding mechanism can lead to a significant improvement in performance.

Finally, several terms are used regarding the performance of the multicomputer and the communication network. *Bandwidth* refers to the amount of traffic a communication medium can carry over a fixed period of time, typically measured in bits per second. The medium may be a single communication link or the entire network as a whole. *Delay* refers to the amount of time which

elapses from when a message/packet enters the communication medium, until it leaves. A more precise definition will be given later. *Latency* is another term which is used interchangeably with delay. Finally, *speedup* refers to the ratio of the execution time of an application program on a single-processor computer system to the corresponding time on a multicomputer system. Intuitively, it indicates how much faster the program executes on the multicomputer.

### 1.3. Previous Work in Communication Networks

The research most applicable to the work reported here may be broadly divided into two categories: loosely coupled computer networks, and interconnection networks for tightly coupled multiprocessors. Each of these will now be discussed in turn.

#### 1.3.1. Loosely Coupled Computer Networks

A great deal of research has been carried out in loosely coupled communication networks. Although many of the constraints and goals in the design of these networks are different from those discussed here, much of this research is still applicable. A complete overview of the literature in this field is beyond the scope of the present discussion. Textbooks such as [Davi73, Tane81, Kuo81, Ahuj82] provide excellent introductions to the field as well as extensive bibliographies. The research most relevant to the communication component networks discussed here deals with message routing techniques and protocols for error free transmission. Other research in relevant areas (e.g. deadlock prevention) will be described later as the need arises.

Message routing is the process of selecting a route, i.e. a path, through a network from a processor sending a message to the processor receiving it. Research in this area is usually concerned with developing general techniques which are applicable to networks of arbitrary topology. An overview and taxon-

omy of practical routing algorithms is described in [McQu74, Gerl81]. Practical routing algorithms used by specific networks have been described for several networks, e.g. Arpanet [McQu74, McQu80], Datapac [Spro81], Tymnet [Tyme81, Rind77], and IBM's SNA [Juen76]. Other heuristic routing schemes have been proposed, among them [Floy82, Fran71, Chou81]. Finally, routing techniques based on more rigorous mathematical performance models include [Cant74, Gall77, Sega77]. Networks constructed from communication components must also use some routing algorithm to establish virtual circuits, so much of the work described above is applicable here.

Another significant area of research in loosely coupled networks is in the design of protocols to ensure reliable transmission of data through the network. A good survey of work in this area and an extensive bibliography is reported in [Pouz78]. Much of the work in protocols has centered around the development of a layered structure of communication protocols, and defining standard protocols within each layer. As a result of this work, a standard has been defined by the International Standards Organization (ISO), and is now widely used by many computer manufacturers [Zimm80].

Of special interest here are protocols for flow control, i.e. mechanisms which control the flow of traffic through the network. Good overviews of work in this area are presented in [Pouz81, Pouz78, Kahn72]. Flow control procedures in Datapac and Tymnet are described in [Spro81, Tyme81, Rind77], while a hierarchical flow control scheme is presented and analyzed in [Chu77].

Most of the protocols developed for loosely coupled networks are inappropriate for the networks discussed here. This is because these protocols make assumptions which are not valid in closely coupled multicomputer networks. In particular, loosely coupled networks cover wide geographic areas and are subject to adverse environmental conditions, so error rates can be expected



to be much higher than in networks constructed from communication components. As a result, protocols in loosely coupled networks typically pay close attention to detecting and retransmitting corrupted messages at *all* levels of the layered structure. With low error rates however, transmission errors can be handled by high-level (i.e. end-to-end) protocols, freeing lower level mechanisms within the network to incorporate such performance improving techniques as virtual cut-through. Thus, the protocols used in loosely coupled networks are normally too inefficient for the networks discussed here.

### 1.3.2. Interconnection Networks for Closely Coupled Multiprocessors

A great deal of research has also been done in the area of interconnection networks for closely coupled multiprocessor systems. "Classical" research in interconnection networks examines single- and multistage-interconnection networks constructed from small (typically 2 by 2) crossbar switches. These networks are discussed in the context of establishing processor to memory or processor to processor communications. The bulk of the remaining research in the field focuses on interconnection topologies. A good survey of work in both of these areas is given in [Feng81].

The work in single- and multiple-stage interconnection networks can be partitioned into two categories: networks for SIMD (single-instruction stream, multiple-data stream) computers, and networks for MIMD (multiple-instruction stream, multiple-data stream) computers. A survey of interconnection networks for SIMD computers is given in [Sieg79a]. Many of the SIMD networks are also applicable to MIMD machines.

SIMD systems are special purpose computers typically used for large computational tasks requiring many vector operations. A "typical" SIMD computer is shown in figure 1.2. Here, a number of processors are connected to memory modules through an interconnection network. The controller broadcasts



instructions to the various processors. All processors execute the same instruction on each clock cycle. Each performs some computation using data from one of the memory modules. If data (e.g. elements of a vector) are properly distributed across the memory modules, then conflicts in accessing the memories can be avoided.

In the scenario described above, the interconnection network effectively *aligns* data scattered across the memory modules. Alternatively, the network can be thought of as performing some permutation of input lines to output lines. Thus, these networks are sometimes referred to as alignment or permutation networks. Networks which support any permutation of input to output lines are

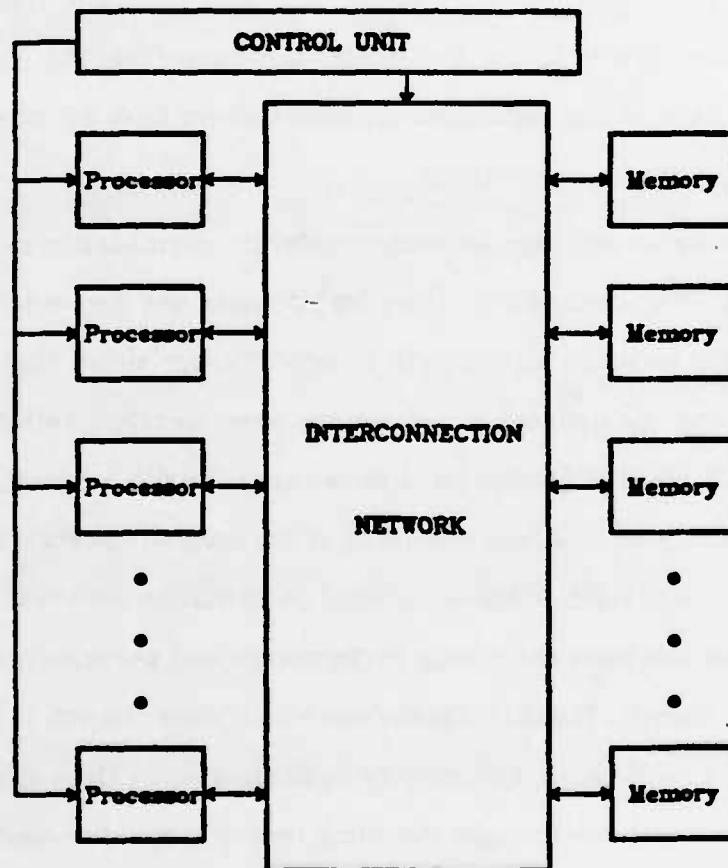


Figure 1.2. A typical SIMD Machine.

called "nonblocking". The crossbar switch [Wulf72] and the Clos network [Clos53] are examples of nonblocking networks. Nonblocking networks become prohibitively expensive as the number of processors and memory modules grows, so less expensive networks which support some subset of all possible permutations (called "blocking" networks) have been explored. Examples of blocking permutation networks include the shuffle exchange [Ston71], banyan networks [Goke73], the omega network [Lawr75], the flip network used in the STARAN processor [Batc76], the indirect binary n-cube [Peas77], the baseline [Wu80a], and the reverse-exchange network [Wu80b]. An introduction and overview of this work is presented in [Chen81]. Classes of networks which subsume many of the specific networks listed above have also been discovered, e.g. the delta network class [Pate81] and the multistage cube [Sieg81]. Thus it is not surprising that many of the variations described above have no, or only slightly different, performance characteristics.

Extensive analyses and comparisons of different permutation networks have been performed. For example, in [Sieg79b] bounds are derived for the time required for some networks to simulate others. Parker shows that the inverse omega network and the indirect binary n-cube have identical switching characteristics [Park80], while in [Wu80a] it is shown that the flip network, omega network, indirect binary n-cube, and one form of the banyan network are topologically isomorphic. Equivalence classes among permutation networks are defined in [Prad80]. Other analyses describing performance and permutation properties include [Fran81, Nass81, Than81]. Extensions which allow the set of performable permutations to be expanded, typically by cascading more than one network or allowing multiple iterations through the same network, are discussed in [Yew81, Wu81a]. A theory for composing the permutations performed by the omega network is discussed in [Ste183]. Finally, parallel algorithms for setting up the

switches in permutation networks are described in [Lev81, Nass82]. Although one disadvantage of the permutation networks described above is that the time complexity to setup an  $n$  input network given some permutation is  $O(n \log n)$  with the fastest known serial algorithms, these papers allow settings to be determined in as little as  $O((\log n)^2)$  time in some situations when  $n$  processors are used to perform the computation.

The topologies of the networks described above can also be applied to networks for MIMD machines. Here however, average message delay and network bandwidth are used as performance measures rather than the number of permutations performed. In this context, it has been shown that most of the networks described above yield virtually the same performance [Pate81].

These interconnection networks represent one class of networks which could be implemented with the communication components described here. Special switches designed specifically for these permutation networks (typically, 2 by 2 crossbar switches) have two *apparent* advantages over the general purpose components proposed in this thesis. First, since the network topology is fixed, they may be optimized for efficient message routing. However, with a virtual circuit transport mechanism (described in chapter 4), message routing is reduced to a single read from a relatively small (a few hundred entries at most) table. In current technology, this can be performed in a single clock cycle, where the clock cycle is determined by the rate at which data can be clocked into a chip, so any advantage derived from optimized routing is minimized.

Second, current implementations of the simple 2 by 2 switches require less circuitry than the components described here. However this difference is largely due to the improved functionality of our communication component, rather than some fundamental increase in complexity. The components described here use more sophisticated buffer management strategies than are

typically used in the 2 by 2 switches, and a microcoded engine is provided for implementing failure recovery protocols. Since the performance of a switching node is limited by I/O bandwidth (i.e. there is some maximum number of pins on each chip and some maximum rate at which each pin can be driven) and since off-chip communications are typically an order of magnitude slower than on-chip speeds [Sequ78], this additional complexity is not detrimental to the clock rate. In addition, general purpose components provide enough flexibility to allow networks to be tailored to the communication needs of the system. For example, more bandwidth could be placed near expected points of congestion, e.g. around the disks. Thus, the communication components described here can achieve at least as much performance as the switching nodes in the permutation networks, if not more, as well as provide additional flexibility to the system designer.

Other research in interconnection networks focuses on defining attractive network topologies. This work can be classified into two categories: networks for special purpose computation, and networks for general purpose computation. Special purpose network topologies are aimed toward achieving an efficient mapping of some class of algorithms onto the network. General purpose networks cannot assume any specific algorithm, so they try to optimize some general criteria for goodness, e.g. average hop count between pairs of nodes.

An introduction to research in special purpose networks designed for efficient execution of specific algorithms is presented in [Gott82]. In [Thom80] a theory of VLSI is introduced and bounds for area/time tradeoffs in implementing VLSI chips for specific computations (e.g. the FFT) are derived. Also, the work in systolic architectures examines two dimensional networks suitable for executing certain numerical algorithms for signal processing and matrix manipulation problems [Kung80, Kung82]. There has also been an extensive amount of work in matching problems to such well known topologies as the perfect shuffle

[Ston71], the mesh [Nass79, Prep83], and tree networks [Deke83, Nath83]. The cube-connected-cycles network is another network which exhibits properties favorable for the efficient implementation of certain parallel algorithms [Prep81].

Much of the work in topologies for general purpose computation focuses on defining networks which achieve some characteristic expected to lead to good performance (e.g. small average hop count). One problem in this domain which has received some attention is the " $(d,k)$  graph problem", in which the goal is to maximize the number of nodes in a graph of degree  $d$ , and diameter  $k$  [Acke65, Frie66, Korn67, Stor70, Toue79, Imas81, Memm82, Amar83]. Other topologies recently proposed for communication networks include ringed trees [Desp78], snowflakes [Fink80], clusters [Wu81b], chordal rings [Arde81],  $C_5$ ' graphs [Farh81], binary trees [Horo81], cube connected cycles [Prep81], hypertrees [Good81], lens networks [Fink81], multiple tree structures [Arde82], and mobius graphs [Lela82a, Lela82b]. Comparisons of some of these structures are reported in [Witt81, Swar82, Reed83]. Finally, other research examines topologies which are attractive for fault tolerance, e.g. [Prad82, Adam82].

Most of this work is directly applicable to the networks studied here, since it refers to topologies which can be constructed from the proposed components. These earlier studies are at a higher level of abstraction than those presented here however. While the work reported above focuses entirely on system performance, the work described here is aimed at low level design decisions, e.g. the number of I/O ports on each chip, and considers the constraints imposed by a VLSI implementation. The impact of these constraints on overall performance is examined. Some work in implementation issues has been performed by Franklin, however this has been restricted to studies of partitioning certain switching structures, e.g. crossbar switches and banyan networks, into modules suitable

for VLSI implementation [Fran82].

In the area of communication components, some building block modules have been proposed. In [Hopp79] a packet switched 2 by 2 crossbar node using unidirectional links is proposed as a switching node. Routing information is carried with each packet as a sequence of bits, with each bit indicating the direction the packet is to follow at intermediate nodes. One disadvantage of this scheme is that the destination address of each node varies according to the location of the sender of the message, and senders are required to generate this routing information themselves. With arbitrary networks, this computation is somewhat complex, and recomputations are necessary if the topology changes because of component failures or network expansion. Simple flow control and buffering scheme are provided, although they do not prevent some of the buffer hogging and deadlock problems discussed in chapter 4. Unlike the design presented here, no processor is provided in each switching node. Overall, this component can be regarded as similar in intent to the components described here, however much less sophisticated in functionality.

A component similar to that described above is the Dual Interconnecting Modular Network Device, or DIMOND [Jans80]. Again, this component has two input and two output links, and each message carries detailed routing information with it. In [Jans80] details of the implementation of the DIMOND are explained, as well as its use in constructing networks such as rings and trees. A minimal amount of buffering is provided in each component (a single register on each output port).

Finally, a 3 input, 3 output link component called STICS (Synchronous Triangular Interprocessor Connection Scheme) has also been proposed [Rile82]. These components can only be applied to a very restricted class of topologies however, and thus are not as general as the components described here.

To the author's knowledge, all of the previous work in VLSI communication components has emphasized simplicity at the expense of generality, functionality, and/or performance. With advances in VLSI however, chip densities are increasing at a rapid rate, and more functionality can readily be integrated onto a single chip. Thus, more sophisticated designs achieving greater functionality and performance are becoming practical. The communication components described here represent one attempt to design and analyze the performance of such a switching chip.

The work presented here is a continuation and extension of the work in the communication switch for the X-tree project [Sequ78, Desp78]. Work in the low-level design of the internal structure of an X-tree node are described in [Laur79, Grif79, Fuji80, Wong81]. Perspectives and lessons learned from these designs and the X-tree project as a whole are described in [Sequ82]. While the work in X-tree focussed on a particular topology, the components proposed here provide more flexibility, allowing construction of arbitrary high-performance communication networks.

#### 1.4. Overview of Thesis

This thesis focuses on the design of VLSI communication components, and the impact of certain design decisions on system performance. The remainder of this thesis is organized as follows: In chapter 2, the tradeoff between the number and bandwidth of the communication links is discussed in the context of a single-chip implementation of the proposed communication component. It is seen that the I/O bandwidth of each component is fixed, and assumed to be equally divided among the attached links. The communication network is modeled as a set of nodes (components) interconnected by communication links. The question of whether each node should have a large number of low bandwidth links (implying relatively few "hops" between a given pair of nodes)



or a small number of high bandwidth links (implying many hops) is addressed. Each node of a topology requiring  $b$  "branches" or links to neighboring nodes can be implemented by a cluster of  $p$ -port communication components.  $M/M/1$  queueing models are used to analyze optimal value of  $p$ , using average delay and total bandwidth of the "cluster node" as performance measures. It is found that components with a small number of ports yield cluster nodes with the most bandwidth, and smallest average delay.

Cluster nodes using components with a small number of ports require more chips than those using a larger number of ports. Thus, the cluster node studies do not consider differing chip counts. Networks with the same number of components are compared within certain classes of network topologies (e.g. lattices and trees). It is found that components using a small number of ports, e.g. from 3 to 5, tend to yield networks with lower average delay, but less bandwidth than networks using components with a larger number of ports. It is argued however, that while network bandwidth can be increased by using more communication chips, average delay cannot be reduced so easily. Thus, components with a small number of ports should be used.

In chapter 3, results of simulation studies are presented. Here, parallel application programs are used to create traffic loads for networks constructed from communication components. The traffic loads cover a wide variety of different communication patterns. Both cluster node networks and networks using approximately the same number of components are examined. The simulation results support the conclusion of the previous chapter that a small number of ports should be used.

Chapter 4 examines the design of a communication component in greater detail, and discusses the complexity of the required circuitry. Various mechanisms for transporting data through any communication network are discussed.

and a mechanism based on virtual circuits is argued to be the most appropriate for the networks discussed here. Schemes for providing hardware support for communication functions — routing, buffer management, and flow control — are presented, and estimates of the number of buffers and virtual channels are determined. Based on these estimates, the amount of circuitry to implement a communication component is estimated, and a floorplan for one implementation is shown.

Finally, chapter 5 presents concluding remarks, and a summary of design recommendations for implementing general purpose, high-performance VLSI communication components.

## CHAPTER TWO

### PERFORMANCE EVALUATION STUDIES

In this chapter, the performance of networks constructed from VLSI communication components is evaluated. The optimal number of communication ports for each chip is discussed in detail. The performance improvement resulting from incorporating a virtual cut-through mechanism into the communication hardware is also studied.

The first section discusses constraints imposed by a single-chip implementation of the communication components. These constraints lead to a tradeoff between the number and bandwidth of the communication links (or ports since there is one link per port). The following section discusses analytical studies evaluating the performance of various networks constructed with VLSI communication components as a function of the number, and thus of the bandwidth of the communication links.

#### 2.1. VLSI Constraints

A VLSI chip is subject to a number of technological constraints. Violation of these constraints will result in a chip which cannot be manufactured in large quantities, or which cannot be depended upon for reliable operation. For this study, the three most important constraints are:

- (1) Limited amount of silicon area.
- (2) Limited allowable power dissipation.
- (3) Limited number of pins for off-chip communications.

We will consider the implications of each of these constraints on the design of a VLSI communication component, and in particular, on the number and

bandwidth of the communication links.

### 2.1.1. Area

Beyond a certain die size, the yield, i.e. the fraction of manufactured chips which function correctly, decreases dramatically with increased area [Glas78]. Current technology allows approximately 500,000 transistors to be placed on a single chip. It is projected that chips with 1,000,000 device will be possible by 1985 [Patt80]. It will be demonstrated in chapter 4 that this is more than adequate for the communication components described here, so limited amounts of silicon area do not severely constrain the design of the chip.

### 2.1.2. Power

The total amount of power generated by the chip must not exceed some upper bound determined by the power dissipation capacity of the integrated circuit package. Since the average power dissipation determines the amount of heat generated by the chip, violation of this constraint will result in a component which will overheat and fail during operation. We will assume that the amount of power dissipated by the chip varies linearly with the number of links, i.e. the total amount of power consumed by a  $p$ -port component is  $(P_p \times p) + C_r$ , where  $P_p$  is the average amount of power consumed by each port, and  $C_r$  is the power dissipated by circuitry which is not affected by the number of ports (e.g. portions of the control and routing circuitry). The power dissipated by this "link independent" circuitry is assumed to be constant; it thus reduces the total amount of power the port circuitry can dissipate, but does not enter into the tradeoff to be discussed.

If, for the moment, we neglect static power dissipation, then the power dissipated by the link circuitry is proportional to the clock rate [Carr72]. Doubling the number of links doubles the amount of circuitry, and thus the power dissi-

pated by the chip. This can be offset by halving the clock rate, which in turn, halves the bandwidth of each communication link. Thus, increasing the number of links requires a proportional decrease in the bandwidth of each one.

Let us now consider static power dissipation. If it is assumed that the static power dissipation of each transistor remains constant as the clock rate is varied, then increasing the number of ports increases the number of transistors, which in turn increases *both* the static and dynamic power dissipation of the chip. However, reducing clock speed only reduces dynamic dissipation. Thus, increasing the number of ports really implies a more than proportional decrease in link speed. Therefore, the linearity assumption is biased to favor a large number of ports.

On the other hand, a slower clock rate implies that smaller transistors may be used, resulting in a reduction in static, as well as dynamic, power dissipation. If the clock rate is cut in half, the current driving capabilities of (say) an NMOS transistor may also be cut in half, which in turn halves the static power dissipation. In other words, both static and dynamic power dissipation are proportional to the clock rate. This is in agreement with the original model which only considered dynamic power dissipation, so link bandwidth is again a linear function of the number of links.

Therefore, when power dissipation is considered, the linearity assumption can only be biased to favor a large number of ports. In the analysis which follows, it will be seen that a small number of ports yields better performance under the linearity assumption. A more complex model which accounts for the bias will only add further support for this conclusion. Here, it will be assumed that link bandwidth is inversely proportional to the number of communication links if power restrictions constrain the design of the chip.

### 2.1.3. Pins

The number of interconnections to the chip's periphery is limited, and will increase much more slowly than the number of transistors per chip [Keye79]. Given  $N$  pins for  $p$  communication links, there are  $N/p$  pins per link. Bandwidth per link is thus proportional to  $N/p$ , assuming a constant bandwidth for each pin. Doubling the number of links halves the number of pins, and thus the total bandwidth, of each link. Thus, due to pin limitations, bandwidth per link also varies inversely with the number of links.

In the analysis presented above, it was assumed that all of the pins of each link are used for transmitting data. In a real implementation, some of the external connections may be used for control lines. These control lines represent an overhead which increases with the number of links. Doubling the number of links doubles the number of control lines, implying fewer pins are available for transmitting data. This results in a more than proportional decrease in link speed. A more precise model which includes control pins will lead to better performance for networks with a small number of ports, since the simplified model described above does not include this "per link" overhead. Again, the more precise model strengthens the conclusions which follow.

Finally, this model neglects the effects of data skew. In a traditional implementation of a parallel communication link, the receiver must wait for all of the arriving bits to reach a stable value before clocking the data. Due to possible variations in propagation delay along the different wires of the link, a parallel link must usually operate at a slower clock rate than the corresponding serial link, an effect not accounted for in the analysis presented above. These data skew problems can be alleviated by implementing the parallel link as a number of autonomous serial links, allowing the link to operate at the highest possible clock rate. This latter implementation leads to link speeds which are propor-

tional to the number of pins per link, in accordance with the linear model presented above.

#### 2.1.4. Summary of Constraints

A tradeoff exists between the number of links per chip and the speed of each link. If the chip design is constrained by either power or pin limitations, then doubling the number of links either halves the clock rate or halves the number of pins allocated to each one. In either case, the link bandwidth is halved. In effect, each chip has some total amount of I/O bandwidth which is equally divided among the existing communication links. This "constant bandwidth per chip" model will be used in all of the studies which follow.

In addition to its effect on link speed, the number of ports also affects the average hop count between two nodes in the network (e.g. a ternary tree could be used instead of a binary tree if one more port were available for each node). As the number of links on each chip is increased, the average hop count between pairs of nodes is reduced. The sections which follow present analytical and simulation results exploring this tradeoff between link speed and hop count.

#### 2.2. Analytical Studies

In this section, the performance of networks constructed from  $p$ -port communication components is evaluated through analytical models. Average "end-to-end" delay and total network bandwidth are used as performance measures. The delay from point A to point B in a network is defined as the time which elapses from when the packet header begins to leave A to when the entire packet arrives at B. The "hop count" from A to B is defined as the length, i.e. the number of links, of the minimum length path from A to B. Network bandwidth is the amount of traffic the network can carry over some fixed time interval. A more precise definition for bandwidth will be given later.



In a real network carrying traffic generated by a parallel application program, average message delay may not be an appropriate performance measure. If a data value is generated in one processor long before it is used by another, then delays encountered by the message carrying this data do not affect the execution time of the program, so long as the data arrives before it is needed. However, since we cannot know a priori which message delays affect performance, *average* delays will be used. Also, averages are simpler to compute than other measures, e.g. maximum delay. A more detailed simulation study will be discussed in chapter 3 which uses execution time (actually, speedup) as the performance measure.

In order to evaluate the tradeoff between hop count and link bandwidth, two multicomputer network models are developed. In the first model, the implementation of a topology requiring  $b$  "branches" per node with  $p$ -port communication components ( $p \leq b$ ) is considered. To achieve the necessary fanout, several components are interconnected to form a "cluster node" with  $b$  external branches. Each cluster node forms a single conceptual node of the desired topology. Delay and bandwidth are compared for various values of  $p$ . In general, a cluster node using components with a small number of ports will require more components than one using a larger number of ports. Thus, comparisons under this model neglect chip count.

A second analysis compares networks using the same number of components. In this model, the hop count/link bandwidth tradeoff is evaluated within individual classes of network topologies, such as trees or lattices.

In each case, a queueing model is used to evaluate network performance. The assumptions made by this model are outlined in the next section. Performance with and without a virtual cut-through mechanism is explored. Delay in a lightly loaded network and overall network bandwidth are computed and com-

pared for the different approaches.

### 2.2.1. Assumptions

As discussed earlier, it is assumed that the bandwidth of each communication link is a linear function of the number of links on each chip. In the analysis which follows, a queueing model is used, and a number of other assumptions must be made:

- (1) Message arrivals at different nodes are independent.
- (2) Message arrival times have a Poisson distribution.
- (3) Message lengths have an exponential distribution.
- (4) Each node contains unlimited buffer space.
- (5) Routing through each node is deterministic.
- (6) Electrical propagation delays are negligible.
- (7) Transmission error rates are negligible.

The first three assumptions are necessary to solve the queueing model. In particular, the first assumption, often referred to as the "independence assumption", states that "the exponential distribution [for message length] is used in generating a new length each time a message is received by a node ..." [Klei76]. This is clearly false since messages maintain their length as they pass through the network, but the effect of the assumption on the accuracy of message delay computations is negligible so long as the network does not contain long chains with no interfering traffic [Kerm79]. The assumption is a reasonable one for the networks examined here because the traffic loads used in these studies lead to output links which carry traffic arriving from several different input links, eliminating the long chains described above.

Similarly, the Poisson arrival time and the exponentially distributed message length assumptions (the latter implies exponential service times) allow the use of  $M/M/1$  queues which can be easily solved. Relaxing each of these assumptions results in  $G/M/1$  and  $M/G/1$  queues respectively. If these queues are used however, Jackson's theorem [Jack57] cannot be applied, since the arrival times at each node no longer follow a Poisson distribution. The resulting queueing models are difficult to solve for the large, complex networks studied here. These assumptions are simplifications since traffic in the actual network need not be Poisson, and the networks considered here use fixed length packets, as will be discussed in chapter 4. Simulation studies will be discussed later which remove these restrictions. Further, a second approximate queueing model using  $M/G/1$  queues will also be discussed [Klei76]. Here, the approximating assumption that Jackson's theorem still applies is made. It will be seen that although this second approximate model yields performance curves somewhat different from the first, the final conclusions drawn from the two models are identical.

The remaining assumptions listed above are appropriate for the networks examined here. The fourth assumption, unlimited buffer space, will be addressed in chapter 4. It will be seen that components with a limited number of buffers can achieve virtually the same performance as components with unlimited buffering capacity. The deterministic routing assumption is appropriate because packets traveling along the same virtual circuit follow the same path from source to destination. As discussed in chapter 4, this is necessary to ensure that packets sent on the same virtual circuit arrive in the order in which they were sent, thus avoiding much of the overhead associated with reassembling messages from their constituent packets. Since communication links are short, electrical propagation delays are negligible (a few nanoseconds at most)

compared to the time required to transmit a single packet (hundreds or thousands of nanoseconds). Finally, the assumption concerning error rates is justified by the extremely low error rates measured in local communication networks [Shoc80]. Since the communication system described here is confined to an even smaller physical area than these local networks, it is less susceptible to noise in the operating environment, making this final assumption even more appropriate.

In addition to the "queueing model assumptions" described above, it is assumed that the internal structure of each cluster node is a balanced tree topology (a tree with minimal average path length between the root and leaf nodes [Knut73]). This minimizes the average hop count through the cluster node, as well as the number of components required to implement a node with a fixed number of branches.

Finally, in order to evaluate the performance of any communication network, traffic distribution assumptions, i.e. which processors send messages to which other processors and how frequently, must be made. These will be explained during the analysis as the need arises. In general, these assumptions are made to simplify the analysis. Simulations using a wide variety of traffic distributions are discussed in chapter 3.

### 2.2.2. Model I: Cluster Nodes

Consider the implementation of a network topology requiring  $b$  branches, i.e. communication links, for each node. Each node could be implemented with a single communication component requiring  $b+1$  ports, assuming one port is used to connect to the computation processor attached to that node. Alternatively, each node could be implemented with a "cluster" of  $p$ -port communication components, where  $3 \leq p \leq b$ . As discussed earlier, it will be assumed that the components within each cluster node are interconnected by a balanced tree

topology. Figure 2.1 for example, shows a node with 4 branches ( $b=4$ ) implemented with 3-port communication components called "Y-components". This "cluster node" implementation implies a larger hop count between processors, however it also uses links of higher bandwidth, since fewer ports are required on each VLSI chip.

Adding a  $p$ -port component to an already existing cluster node adds  $p-2$  branches. Since the one component cluster node has  $p-1$  branches, an  $n$  component cluster node has  $(p-1) + (p-2)(n-1)$  branches. Thus, a  $b$ -branch cluster node uses

$$n = \text{ceiling} \left[ \frac{b - p + 1}{p - 2} + 1 \right] = \text{ceiling} \frac{b - 1}{p - 2}$$

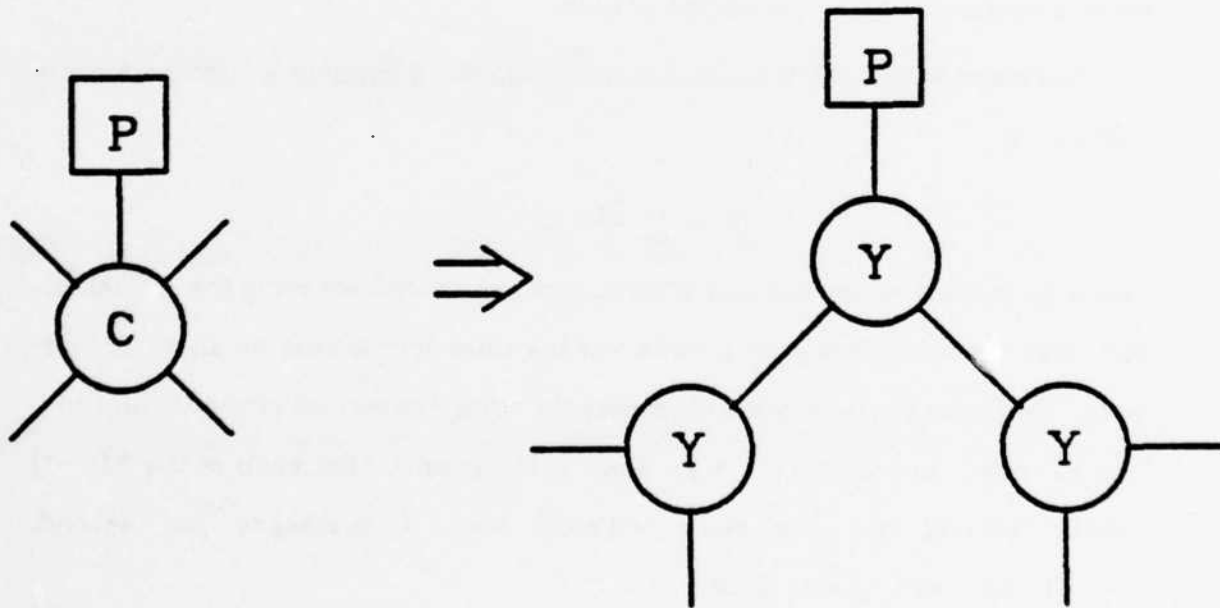


Figure 2.1. 4-branch node built from Y-components.

components, where  $\text{ceiling}(x)$  is defined as the smallest integer greater than or equal to  $x$ .

### 2.2.2.1. Queueing Model

The queueing model presented in [Klei76] is used to evaluate the performance of a  $b$ -branch "cluster node". In order to evaluate these models, traffic distribution assumptions must be made. For the cluster node network, it is assumed that there are two virtual circuits between every pair of branches in the cluster node, one in each direction. In order to simplify the analysis, traffic to and from the processors attached to the cluster node will be ignored, and only traffic between branches will be considered. Since there are  $b$  branches, there are  $b(b-1)$  virtual circuits through such a node. Assume that a traffic load of  $l$  messages per second exists on each of these virtual circuits, and each message consists of a single packet of data.

The average delay  $T$  through a store and forward communication network is defined as:

$$T = \sum_{i,j} \frac{\gamma_{ij}}{\gamma} Z_{ij}$$

where  $\gamma_{ij}$  is the average number of messages per second entering the virtual circuit from branch  $i$  to branch  $j$ , while  $\gamma$  is the total arrival rate on all virtual circuits.  $Z_{ij}$  is the average delay for messages along the virtual circuit from  $i$  to  $j$ . It is assumed that  $\gamma_{ij} = 0$  if  $i = j$ . Since it is assumed that each of the  $b(b-1)$  virtual circuits has the same external load,  $l$  messages per second,  $\gamma = b(b-1)l$ , and  $\gamma_{ij} = l$ . Thus,

$$T = \frac{1}{b(b-1)} \sum_{i,j} Z_{ij} \quad (1)$$

Let us now examine  $Z_{ij}$ .

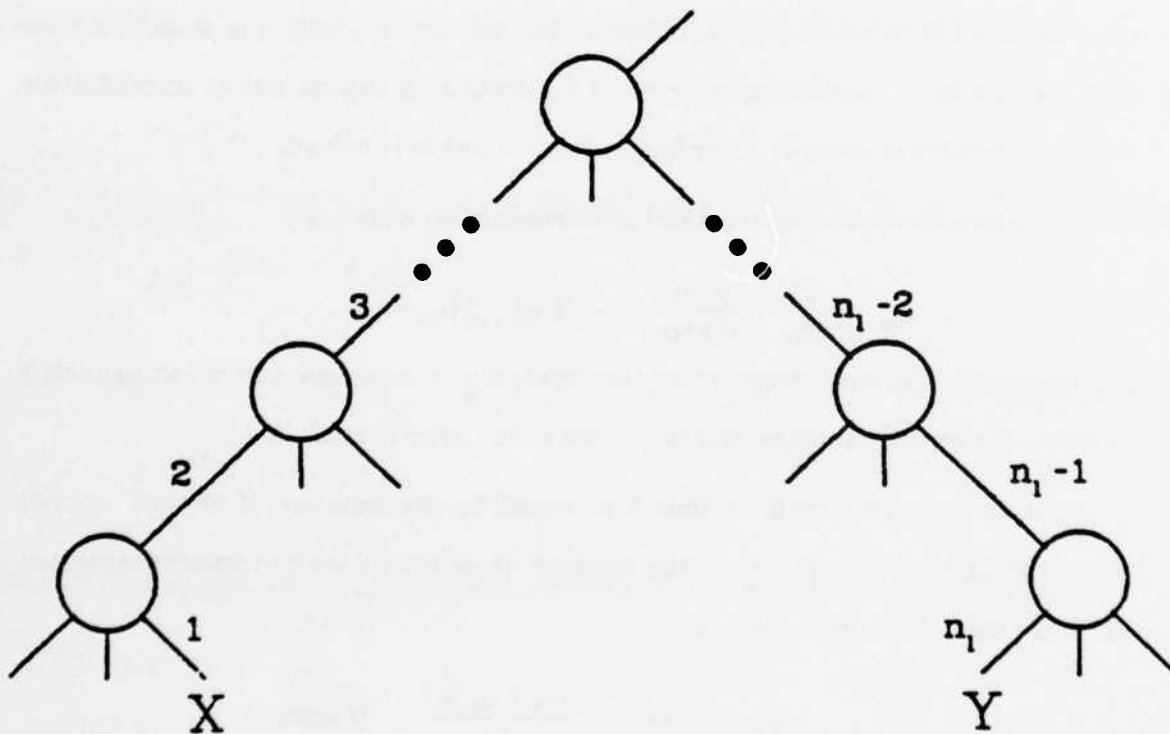
Consider the path taken by the virtual circuit from branch  $X$  to branch  $Y$ , as shown in figure 2.2. The average delay  $Z_{xy}$  along this path is equal to

$$Z_{xy} = \sum_{i=1}^{n_l} T_i$$

where  $T_i$  is the average delay at link  $i$ . Assume links are numbered sequentially from 1 to  $n_l$ , as shown in figure 2.2. With cut-through,

$$T_i = \frac{m_i}{C(1-\rho_i)} - (1-\rho_{i+1})(t_m - t_h) \quad (2)$$

as discussed below and in [Kerm79], where



**Figure 2.2.** *Virtual circuit from X to Y.*



- $m_i$  = average message length
- $C$  = capacity (bandwidth) of the links
- $\rho_i$  = utilization of link  $i$
- $t_h$  = time to transmit message header over the link
- $t_m$  = time to transmit message over the link

The message transmission time,  $t_m$ , includes the time to send both the data and header portions of the message. Assuming the total I/O bandwidth of each  $p$ -port component is  $B$  bits per second,  $C$  is equal to  $B/p$ . The first term of equation (2) is the solution of an M/M/1 queueing model, and represents the amount of time required to obtain and transmit a message over the link. The second term considers the effect of cut-through.  $(1 - \rho_{i+1})$  is the probability that a cut-through occurs, and  $t_m - t_h$  is the amount of time "saved" by beginning to forward the message as soon as the header has arrived. It is assumed that no "partial" cut-throughs occur, i.e. forwarding begins either immediately after the header arrives or after the entire packet is received.

Thus the delay through the virtual circuit from X to Y is

$$Z_{xy} = \sum_{i=1}^{n_i} \left[ \frac{p m_i}{B(1-\rho_i)} - (1-\rho_{i+1})(t_m - t_h) \right].$$

$Z_{xy}$  measures the time from when the header of a message arrives at branch X to when the entire message has been forwarded over branch Y.

The total traffic load on link  $k$  is equal to the number of virtual circuits using the link, say  $v_k$ , times  $l$ , the load on each virtual circuit in messages per second. Thus, link utilization is

$$\begin{aligned} \rho_k &= \frac{m_i l v_k p}{B} & \text{if } k \leq n_i \\ &= 1 & \text{if } k = n_i + 1 \end{aligned} \quad (3)$$

Assigning  $\rho_{n_i+1}$  to 1 forces the  $(1-\rho_{i+1})(t_m - t_h)$  portion of the last term in the summation for  $Z_{xy}$  to be zero. This is necessary to fulfill the definition for delay given above, i.e. the time which transpires from when the head of the message

enters the cluster node until the time at which the *end* of the message leaves. The equation for  $Z_{xy}$  measures the time from when the head of the packet enters until the time at which the *head* begins to leave. Thus we must also add the time which elapses until the *end* of the packet leaves the cluster node. Setting  $\rho_{n_i+1}$  to 1 accomplishes this by in effect, eliminating the "saved time" resulting from cut-through in the final node.

Since  $v_k$  can be easily computed for each link of a given cluster node, the delay  $Z_{ij}$  for each virtual circuit can be found. Once  $Z_{ij}$  is known, equation (1) can be used to compute the average delay among all virtual circuits using the cluster node. Figure 2.3 shows the results of this computation for a 20-branch ( $b = 20$ ) cluster node. The optimal number of ports as a function of  $b$  will be studied in a later section.

The various curves correspond to implementations that differ in two respects:

- (1) the number of ports on each communication component
- (2) whether or not a cut-through mechanism is used

The "without cut-through" curves are obtained by deleting the  $(1 - \rho_{i+1})(t_m - t_h)$  term in equation (2). Average delay is plotted in figure 2.3 as a function of the external load applied to each virtual circuit.

The computations assume that the average packet length  $m_i$  is 17 bytes, consisting of 16 data bytes and a one byte header. The total I/O bandwidth of each chip,  $B$ , is assumed to be 100 Mbits/chip-second, and is equally divided among the existing links. This latter value was chosen arbitrarily but does not affect the relative ordering of the curves. These numerical values will be used in all subsequent computations unless indicated otherwise. From figure 2.3, it is seen that network performance deteriorates as  $p$  is increased for this particular

cluster node.

To a first order approximation, each of the curves in figure 2.3 can be represented by two performance measures:

- (1)  $T^*$ , the delay in a lightly loaded network.
- (2)  $l^*$ , the maximum traffic load the network can support.

$T^*$  is the delay when  $l$ , the traffic load on each virtual circuit, is zero, and  $l^*$  is the asymptotic value for traffic load at which the delay approaches infinity. This latter quantity reflects the point at which some link(s) in the network approach 100% utilization, leading (mathematically) to queues which become infinitely long. In the real network, a flow control mechanism limits the actual queue size on each link, as will be discussed later. We will now examine delay and throughput in turn to determine the optimal number of ports for implementing

**CLUSTER NODE: BANDWIDTH and DELAY**  
(20 branches)

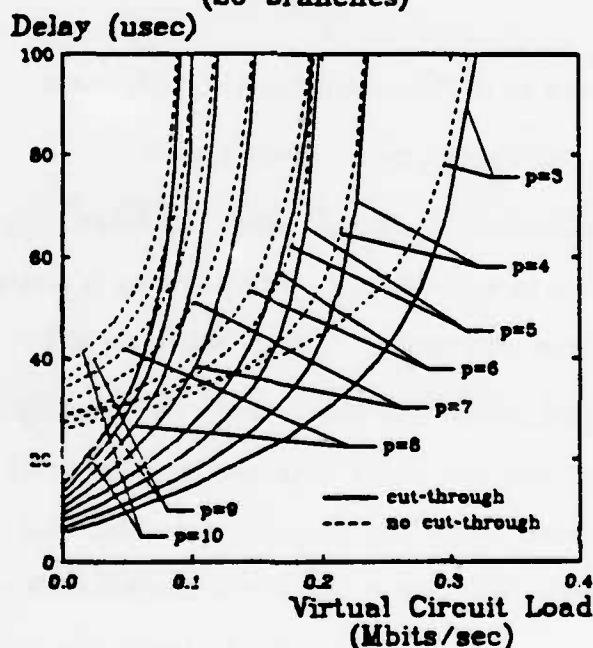


Figure 2.3. Queueing delay for 20-branch cluster node.

cluster nodes of any size.

### 2.2.2.2. Delay

$T^*$ , the delay through a lightly loaded cluster node, is obtained by setting the traffic load,  $l$ , or equivalently the link utilization,  $\rho_i$ , equal to 0 (except if  $i=n_i+1$ , in which case  $\rho_i=1$ ). Thus, from equation (2), the delay at each hop is

$$\begin{aligned} T_i^* &= \frac{p m_i}{B} - (t_m - t_h) & \text{if } i \leq n_i \\ &= \frac{p m_i}{B} & \text{if } i = n_i + 1 \end{aligned}$$

when cut-through is used. A graph of  $T^*$  as the number of branches increases is shown in figure 2.4. It is seen that cluster nodes implemented using components with the minimum number of ports yield the smallest delay. The "bumps" in figure 2.4 occur when a new component is added to the cluster node as the number of branches is increased. This leads to a discontinuity in the average hop count, which in turn causes a discontinuity in the delay.

Without cut-through, the delay of each hop through a lightly loaded  $b$ -branch cluster node implemented with  $p$ -port communication components is simply  $p m_i / B$ . If  $\bar{H}$  is the average number of hops through the cluster node, then  $T^* = \bar{H} m_i p / B$ . This function is also plotted in figure 2.4. It is seen that the optimal number of ports is again never larger than 4. The curves also demonstrate that virtual cut-through can significantly improve message delays.

Assuming the cluster node is implemented by a balanced  $(p-1)$ -ary tree, at most  $2 \log_{p-1} b$  hops are required. Thus, the delay through a cluster node without cut-through is

$$T^* = \frac{2 m_i p \log_{p-1} b}{B}.$$

Differentiating with respect to  $p$  and setting the result equal to 0 reveals that minimum delay is achieved with approximately 4.6 ports per component,

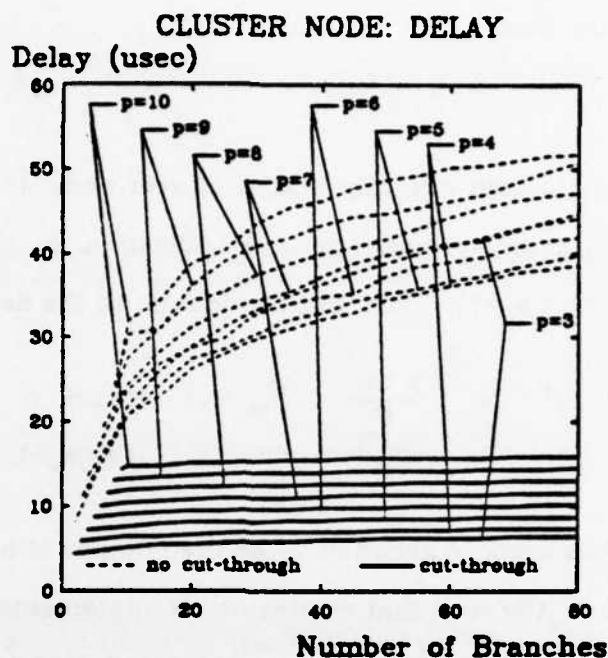


Figure 2.4. Delay through cluster node under light traffic loads.

agreeing with the curves in figure 2.4.

Thus we see that the optimal number of ports is relatively small when considering delay through lightly loaded networks. Delay through a lightly loaded cluster node is minimized when the number of ports is between 3 and 5.

### 2.2.2.3. Bandwidth

$L^*$  is defined as the total network load when  $\rho_i$  approaches 1 on the most heavily utilized link in the cluster node. Since the links around the root of the cluster node carry the most virtual circuits, they will saturate first. If the load on each virtual circuit at saturation is  $l^*$ , then  $L^*$  is  $b(b-1)l^*$ . If the most heavily utilized link has bandwidth  $B/p$  and carries  $v$  virtual circuits, then equation (3) suggests that saturation occurs at  $l^*m_i = B/vp$  bits per second. Thus,

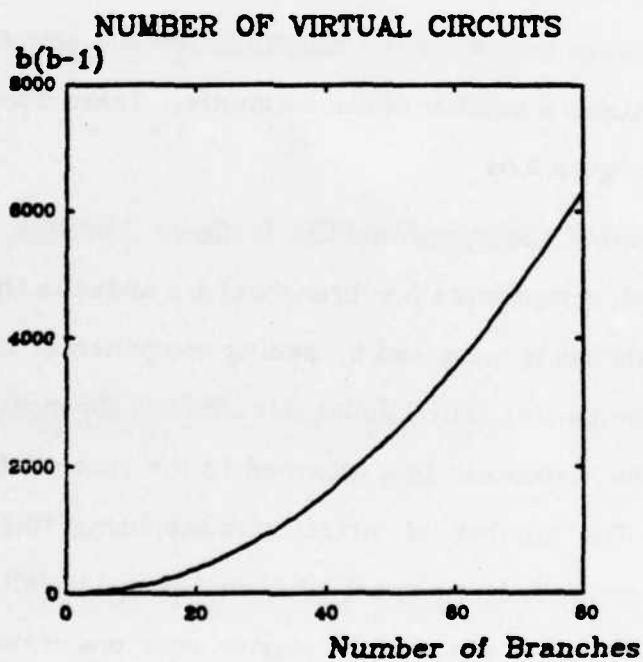
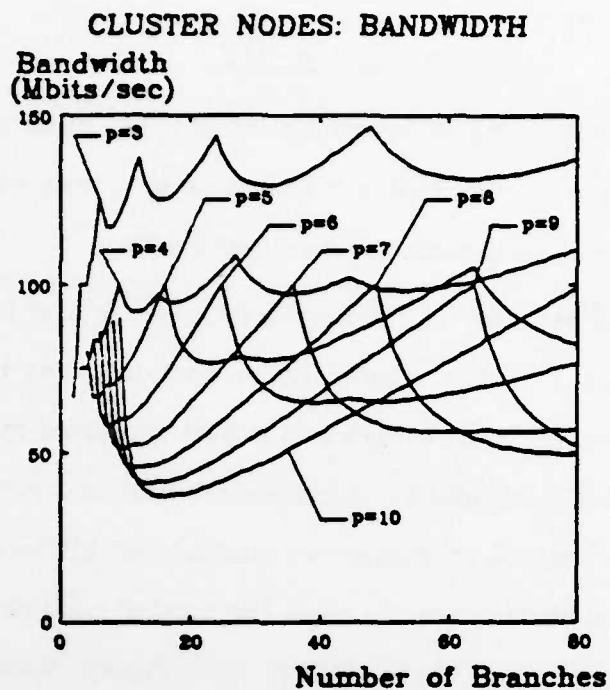
$$L^*m_i = \frac{b(b-1)B}{v p}$$

bits per second. A plot of  $L^*m_i$  as a function of the number of branches is shown in figure 2.5a. The curves indicate that cluster nodes constructed with the minimum number of ports yield the most bandwidth.

The irregular behavior of the curves is an artifact of the manner in which branches are added to the cluster node, and does not represent a general behavior of communication networks. It is best explained by examining the individual components from which the curves are derived. For a given value of  $p$ , the behavior of  $L^*$  can be characterized qualitatively by the quantity  $b(b-1)/v$ , i.e. the number of virtual circuits using the cluster node divided by the number of circuits using the most heavily loaded link. Figures 2.5b and 2.5c show plots of these two quantities as a function of  $b$ . For clarity, only curves for  $p$  equal to 3, 4, and 5 are shown in figure 2.5c. The remaining curves demonstrate a similar behavior. It is seen that while the function  $b(b-1)$  yields a smooth curve, the curve for  $v$  contains a number of discontinuities. These discontinuities give rise to the peaks in figure 2.5a.

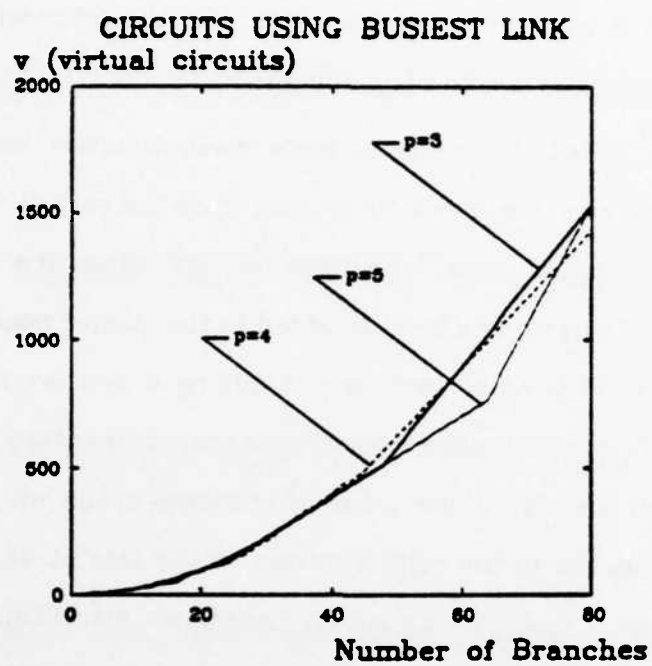
The location of the discontinuities in figure 2.5c is a consequence of the manner in which components (i.e. branches) are added to the cluster node. The number of branches is increased by adding components "from left-to-right" at the leaves of the cluster node. Under this scheme, the most heavily utilized link is always be the "leftmost" link attached to the root of the cluster node (see figure 2.5d). The number of virtual circuits using this link,  $v$ , is simply  $b_1(b - b_1)$ , where  $b_1$  is the number of branches in the leftmost subtree of the root. If a new branch is added to the cluster node, one of two situations occurs:

- (1) The branch is added to the leftmost subtree, causing both  $b$  and  $b_1$  to increase by 1, and  $v$  to increase by  $(b - b_1)$ .

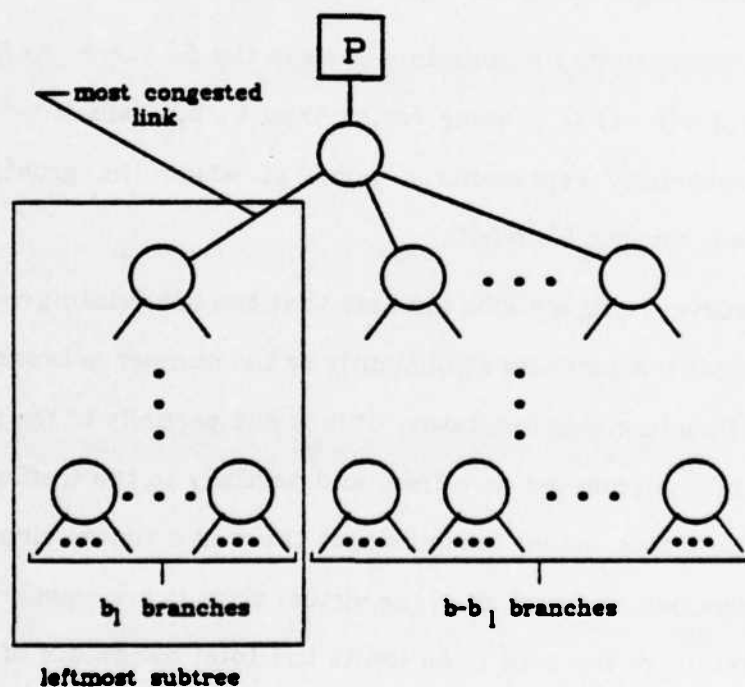


**Figure 2.5.** (a) Bandwidth of cluster node. (b) Circuits in cluster node.





(c)



(d)

**Figure 2.5.** (c) Most heavily loaded link. (d) Sample cluster node.

- (2) The branch is added somewhere other than the leftmost subtree, causing only  $b$  increases by 1, and  $v$  to increase by  $b_l$ .

As branches are added to the cluster node, discontinuities occur when the transition is made between these two situations, since the rate at which  $v$  is increasing suddenly changes. This transition occurs when the leftmost subtree becomes full, and when a new level is added to the cluster node. An exception to this rule occurs for  $p$  equal to 3, where adding a new level does *not* cause a discontinuity. This is a consequence of the symmetric nature of the binary tree. When a new level is added, the number of branches in the left subtree  $b_l$  is equal to  $b - b_l$ , the number in the right subtree, so the rate at which  $v$  is increasing remains the same, and the transition causes no discontinuity. Thus, in the binary tree, discontinuities occur only when the left subtree becomes full, and new branches begin filling the right subtree.

Each discontinuity results in a peak in the  $L^*$  curve. As  $b$  is increased,  $L^*$  increases if  $b(b-1)$  is growing faster than  $v$ , but falls if  $v$  is growing faster. Each discontinuity represents a point at which the growth of  $v$  becomes accelerated, causing  $L^*$  to fall.

The curves in figure 2.5a indicate that the bandwidth provided by the cluster node does not increase significantly as the number of branches, and thus the number of components increases. This is due partially to the fact that the cluster node is implemented as a tree, and partially to the traffic model presented above. The traffic model assumes that there is a virtual circuit between every pair of branches, and that all of the virtual circuits are equally loaded. Thus, the I/O bandwidth of the root node limits the total bandwidth of the cluster node; increasing the number of components does not significantly increase the total bandwidth provided by the cluster node.

When the root node links become congested, most of the links of the cluster node, i.e. those near the leaves, are underutilized. Thus, virtual circuits which only use these links, i.e. circuits which do not go through the root node, can actually handle much more traffic. Let us consider the total bandwidth of the cluster node when traffic on these "underutilized" virtual circuits is allowed to increase. In particular, let us uniformly increase the traffic load on all virtual circuits which do not go through the root node until more links begin to saturate. The links "highest" in the cluster node tree will saturate first. Now repeat this process, i.e. increase the load on all virtual circuits which do not use saturated links, until all of the links are saturated. The total load on all of the virtual circuits gives the maximum traffic load the cluster node will support.

With the traffic load just described, it is clear that all of the links of the cluster node will be equally utilized. Such a network is said to be "balanced". The bandwidth of a balanced network is equal to the sum of the bandwidths of all of the communication links divided by the average hop count through the network. Intuitively, each link adds some fixed amount bandwidth to the network, and each virtual circuit uses bandwidth proportional to the number of hops it requires. Thus, this figure is indicative of the number of active (i.e. transmitting data) virtual circuits the network can support at one time, or alternatively, it is indicative of the total bandwidth allocated to a fixed set of virtual circuits. It will be seen later that this intuitive measure of bandwidth can also be derived from a queueing model for balanced networks.

A  $b$ -branch cluster node built from  $p$ -port communication components provides bandwidth (see section 2.2.2):

$$\frac{B \text{ ceiling } \left\lceil \frac{b-1}{p-2} \right\rceil}{H} \text{ for } p \geq 3.$$

A graph of this measure of bandwidth for various values of  $p$  is shown in figure

2.6. Since a cluster node of  $n$  chips has a total link bandwidth which increases linearly with  $n$ , and the hop count increases only logarithmically in  $n$  (assuming a tree topology for the cluster node), one would expect the cluster node with the most chips to provide the most bandwidth. This corresponds to cluster nodes constructed with components using the minimum number of ports, or here, 3. The graphs confirm this intuitive result. Note that virtual cut-through does not impact the bandwidth provided by a network.

When constructing multicomputer systems with cluster nodes, congestion at the root can usually be alleviated through the use of an appropriate routing algorithm. For example, figure 2.7 shows a grid topology implemented with Y-components. An appropriate routing algorithm for this topology is to route packets along one direction, say north/south, and then the other, east/west,

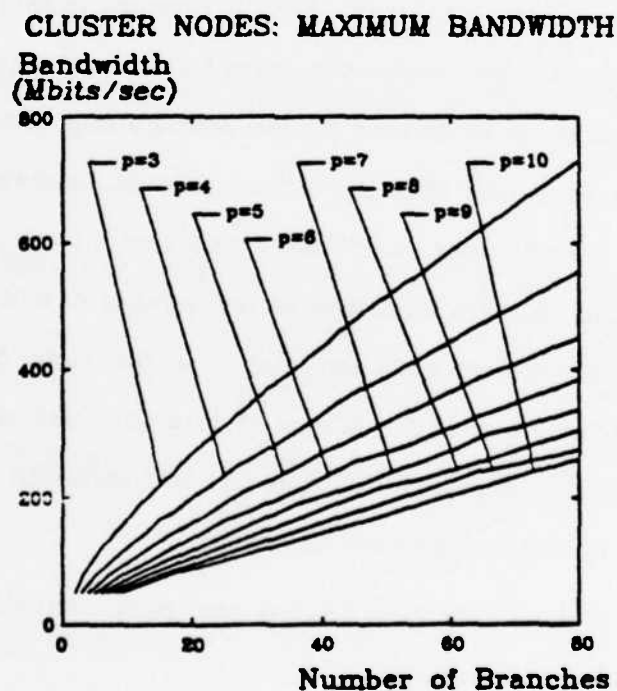
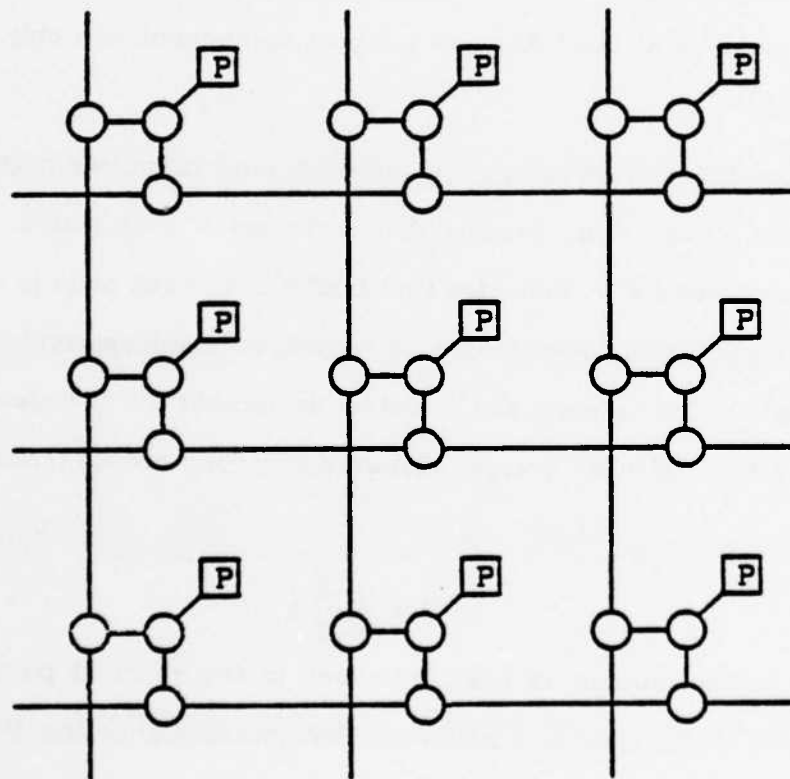


Figure 2.6. Maximum bandwidth (#chips /hop count) of cluster node.

using only one "90 degree turn". With this scheme, each packet travels through the root of a cluster node at most three times — at the source node, at the destination node, and at the node in which the 90 degree turn is made. In general, this type of behavior can be exploited for any topology (except trees where there is only one path between any pair of processors) by using a "global" shortest path routing algorithm through the network to increase usage of the shorter paths through the cluster node which do not go through the root.

Thus cluster nodes built with communication components with a small number of ports, say from 3 to 5, yield the least delay, and cluster nodes built with 3-port components yield the most bandwidth.



**Figure 2.7.** *Grid topology built with Y-components*

### 2.2.3. Model II: Networks with a Fixed Number of Components

The models in the previous section demonstrated that higher bandwidth and lower delays can be achieved by implementing  $b$ -branch cluster nodes with communication components using relatively few ports. Such networks require more chips than networks constructed from components with a larger number of ports. In this section, we explore the tradeoff between hop count and link bandwidth for networks with the same number of switching components.

Consider a large, unbounded network constructed from  $p$ -port communication components. As before, assume that each port has a bandwidth proportional to  $1/p$ . It will be assumed that there is one processor attached to each communication component in the network, using one of its  $p$  ports. Thus, in this model,  $p$  must be at least 4, since a 3-port component can only implement a ring topology.

Suppose that for some application, each node must communicate with all nodes within  $R$  hops of it. Assume that there are  $M$  such nodes. A small value for  $R$ , or equivalently  $M$ , indicates that traffic from each node is very localized, while a larger value indicates more global communications. Consider one specific node in the network, say  $X$ , and let us number the  $M$  nodes it sends messages to:  $1, 2, \dots, M$ . The average distance (i. e. hop count) from  $X$  to these  $M$  nodes is

$$H = \frac{1}{M} \sum_{i=1}^M d_i$$

where  $d_i$  is the number of links traversed in the shortest path from  $X$  to  $i$ . Assume that traffic from  $X$  is uniformly distributed among the  $M$  nodes it communicates with. As before, increasing  $p$  will reduce the average distance, but at the cost of slower links. Conversely, reducing  $p$  implies faster links, but longer distances.

The average distance  $\bar{H}$  is clearly dependent on the topology of the network. In general, more redundancy (i. e. distinct paths between pairs of nodes) implies a larger  $\bar{H}$ , assuming constant  $p$ . For the purposes of this section, different classes of network topologies will be characterized by a function,  $m(i)$  ( $i=1,2,\dots,k$ ), with  $M=\sum_{i=1}^k m(i)$  and  $m(i)$  equal to the number of nodes whose minimum length path to node  $X$  is exactly  $i$  hops. The networks discussed here are assumed to be symmetric and unbounded. Since each node has  $p-1$  ports for communicating with other nodes (one port leads to the processor attached to that node),  $m(1) = p-1$ . Two abstract cases will be discussed here:

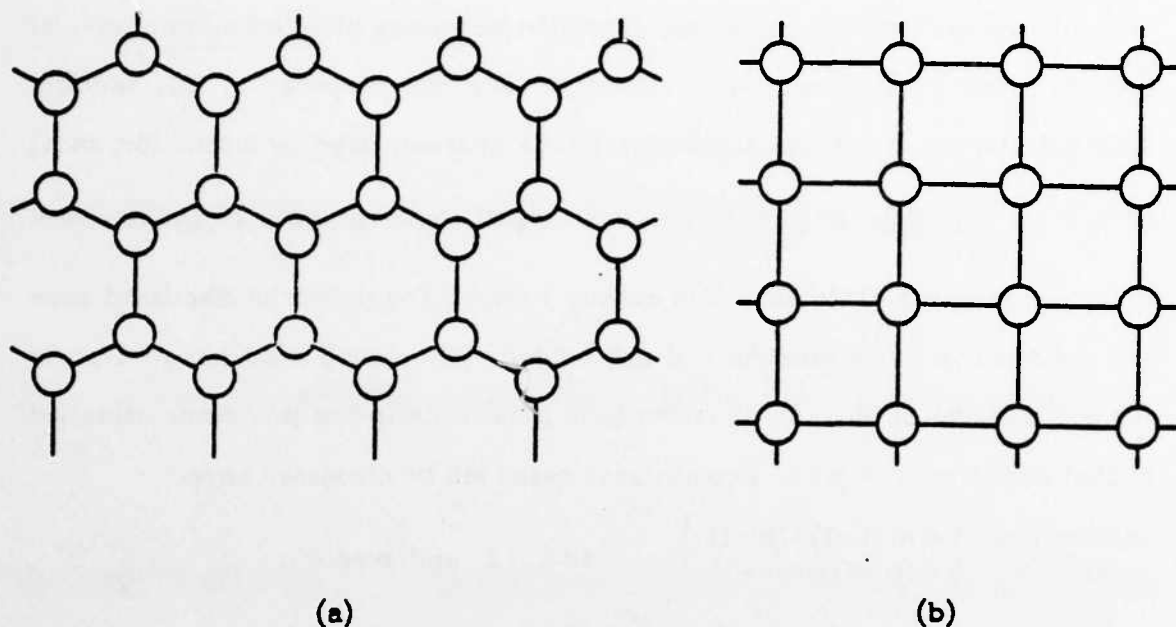
$$\left. \begin{array}{l} \text{lattices: } m(i) = m(i-1) + (p-1) \\ \text{trees: } m(i) = (p-2) \times m(i-1) \end{array} \right\} \quad i=2,\dots,k \text{ and } p \geq 4$$

The first represents regular two-dimensional lattices (see figure 2.8) and the second trees. Note that the latter case has no redundant links and thus gives minimal  $\bar{H}$  for any topology with  $p$  ports per node. It is thus a favorable topology for components with a large number of ports, since networks using these components depend on small hop counts to overcome the handicap of having slower links. The lattice networks represent an alternative class of topologies with less favorable hop count averages, but redundant paths between pairs of nodes.

### 2.2.3.1. Queueing Model

The cut-through queueing model discussed earlier can also be applied to the networks presented in this section. The symmetric nature of the traffic load and the network topology leads to links which are equally loaded, i.e. the network is balanced. As before, we will consider only traffic within the network itself. Delays on the links between the processors and communication components are ignored.





**Figure 2.8.** Two regular two-dimensional lattices. (a)  $p=4$ . (b)  $p=5$ .

A closed form solution for estimating network delay, including the effects of virtual cut-through, is known [Kerm79]. Using the same assumptions discussed in section 2.2.1, it can be shown that the average delay to send a message through a balanced network is

$$T = \frac{m_i p \bar{H}}{B (1-\rho)} - (\bar{H}-1) (1-\rho) (t_m - t_h) \quad (4)$$

where:

- $\bar{H}$  = average hop count
- $m_i$  = average message length
- $B$  = total I/O bandwidth of each communication component
- $p$  = number of ports
- $\rho$  = utilization of each link
- $t_h$  = time to transmit message header over the link
- $t_m$  = time to transmit message over the link

The first term of this equation is the delay when no cut-through is used. The second term is the improvement when cut-through is added. The effectiveness

of cut-through in reducing delay increases with  $\bar{H}$  because there are more chances for cut-through to occur if the number of hops required is large. The dependence on  $\rho$  arises from the fact that the probability that the outgoing link is free, i.e. the probability that a cut-through will occur, depends on how heavily the link is utilized. As before, the model assumes that no "partial" cut-throughs occur, i.e. forwarding begins either immediately after the header arrives or after the entire packet is received. The cut-through mechanism has greater impact in lightly loaded networks (small  $\rho$ ).

Consider a network with  $N$  processors (and thus  $N$  communication components), with each processor sending messages to the  $M$  processors closest to it. If  $\bar{H}$  is the average hop count to reach another processor, then there are  $N \times M$  virtual circuits, each using  $\bar{H}$  links. Assume the load on each virtual circuit is  $l$  messages per second, or  $l m_4$  bits per second. Since the network has  $N \times (p-1)$  links (excluding the one connecting to the processor), the average load on each link is  $N M m_4 l \bar{H} / N(p-1)$  bits per second. Therefore,

$$\rho = \frac{M m_4 l \bar{H}}{B} \frac{p}{p-1}. \quad (5)$$

Since  $\bar{H}$  can be computed numerically, given  $M$  and  $p$ , we can use equations (4) and (5) to compute message delays.

Figure 2.9a shows delay in lattice topologies with and without a cut-through mechanism as a function of the load applied to each virtual circuit. Table 2.1 lists the number of virtual circuits using each link.  $M$  is fixed at 50 nodes. The optimal number of ports as a function of  $M$  will be studied in a later section. Under light traffic loads, networks with a smaller number of ports achieve lower delays, regardless of whether or not a cut-through mechanism is used. Figure 2.9a indicates however, that the "knee" for curves with a large number of ports is further to the right than that of those with a small number of ports. This indicates that networks with a large number of ports can maintain reasonable

delays for larger traffic loads than networks with a small number of ports. In other words, these curves indicate that components with a small number of links yield networks with shorter delay, but less overall bandwidth.

Table 2.1.  
Link Usage

$p$	Link Bandwidth (Mbits/sec)	Circuits per Link (Lattices)	Circuits per Link (Trees)
4	25.00	65.00	57.33
5	20.00	42.50	32.50
6	16.67	30.00	24.00
7	14.29	23.33	18.00
8	12.50	18.57	13.43
9	11.11	15.00	11.50
10	10.00	12.67	10.11

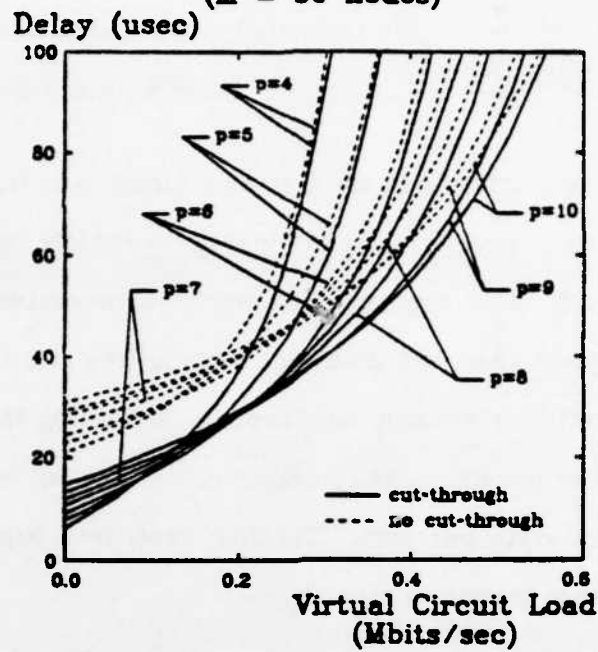
Figure 2.9b and table 2.1 present the same analysis for tree topology networks, also with  $M$  fixed at 50 nodes. Again, networks with a small number of ports yield better delay under light traffic loads, but poorer overall bandwidth. The minimum number of ports achieves the least delay when a cut-through mechanism is used, as would be expected since cut-through diminishes the penalties of traversing extra hops. Networks without cut-through achieve minimal delay when 5 ports are used, for this particular value of  $M$ .

We will now analyze the optimal number of ports as a function of traffic locality, or here,  $M$ . As before,  $T^*$ , the delay in a lightly loaded network, and  $l^*$ , the maximum virtual circuit traffic load supported by the network, will be evaluated and compared.

### 2.2.3.2. Delay

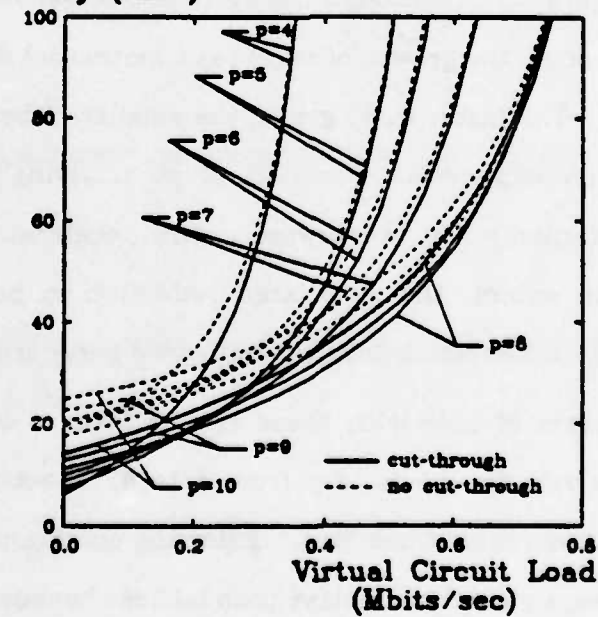
$T^*$ , the delay in a lightly loaded network is again found by setting  $\rho$  equal to 0. Thus, from equation (4), one obtains:

LATTICES: BANDWIDTH and DELAY  
( $M = 50$  nodes)



(a)

TREES: BANDWIDTH and DELAY  
( $M = 50$  nodes)



(b)

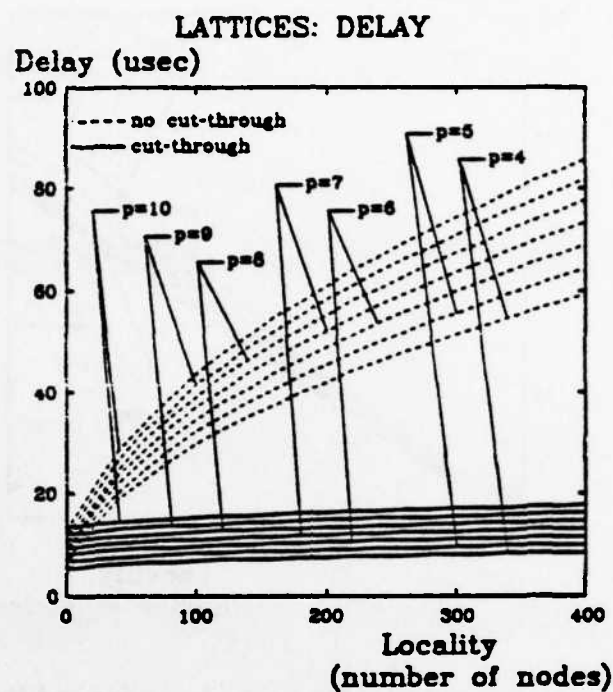
Figure 2.9. Queuing delay,  $M=50$ . (a) lattices. (b) trees.

$$\begin{aligned}
 T^* &= \frac{m_1 p \bar{H}}{B} - (\bar{H}-1)(t_m - t_h) && \text{with cut-through} \\
 T^* &= \frac{m_1 p \bar{H}}{B} && \text{without cut-through}
 \end{aligned}$$

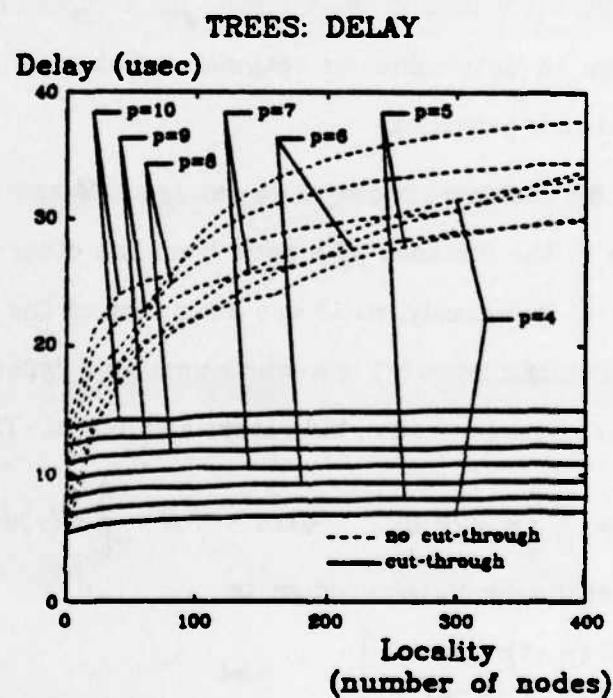
These quantities are plotted in figure 2.10 as a function of  $M$ , the number of processors to which each processor sends messages (which determines  $\bar{H}$ ). When cut-through is used, it is seen that networks constructed with the smallest number of ports yield the least delay for both lattice and tree topologies. The same is true for lattices without cut-through, indicating that the reduction in hop count caused by increasing the number of ports is not enough to adequately offset the lost bandwidth per port. The final case, tree topologies without cut-through, is somewhat more complex.

In tree topologies without cut-through (figure 2.10b) it is seen that the smallest number of ports ( $p=4$ ) does not give minimum delay beyond  $M=32$  nodes. Similarly, as  $M$  is increased further, larger values of  $p$  appear more attractive (see figure 2.11), although the optimal number never rises beyond 6. Given some value of  $M$ , the growth of  $m(i)$  (as  $i$  increases) determines the average hop count,  $\bar{H}$ . The faster  $m(i)$  grows, the smaller  $\bar{H}$  becomes. In tree networks,  $m(i)$  is an exponential function of  $p$ , implying its growth will be accelerated substantially if  $p$  is increased. This acceleration is so substantial that, to a certain extent, the associated reduction in hop count effectively offsets the bandwidth loss which results when more ports are used.

For both classes of networks, these results favor a communication component with relatively few ports, say from 4 to 6. A cut-through mechanism makes the optimal number closer to 4. Under the conditions stated above, tree topologies will always yield lower delays than lattices because of lower hop count averages. This in turn results from the lack of redundant paths in tree topologies and is in agreement with results already discovered by other researchers



(a)



(b)

**Figure 2.10.** Delay under light traffic loads. (a) lattices. (b) trees.

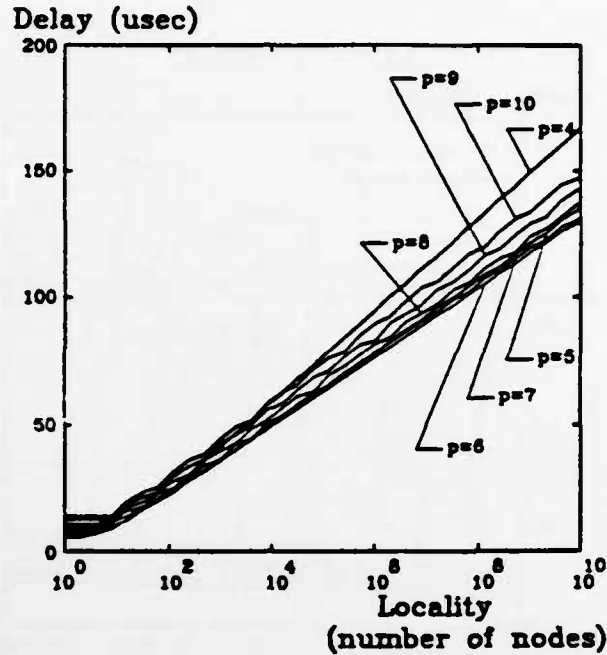


Figure 2.11. Delay under light traffic loads, trees (no cut-through).

[Desp78]. Asymptotic values of  $m_i p \bar{H} / B$  as a function of  $M$  will now be derived, in order to determine an optimum number of ports when no cut-through mechanism is provided.

Given such an abstract topology, we can treat  $M$  as a function of the continuous variable  $\tau$ , the distance of a node from the other nodes to which it is sending messages, (previously,  $m(i)$  was a function of the discrete variable  $i$ ). As  $M$  grows toward infinity,  $m(\tau)$  is asymptotically equivalent to  $m(i)$ . With this perspective,  $\bar{H}$  is no longer a sum, but rather an integral. Thus we have

$$\bar{H} = \frac{1}{M} \int_0^R \tau m(\tau) d\tau \quad \text{with} \quad M = \int_0^R m(\tau) d\tau$$

And the two cases discussed above reduce to:

$$\left. \begin{array}{l} \text{lattices: } m(\tau) = (p-1)\tau \\ \text{trees: } m(\tau) = (p-1)(p-2)^{\tau-1} \end{array} \right\} \quad p \geq 4$$

Evaluation of the above integrals for the two cases results in the following equations for delay:



$$\text{lattices: } T^* = \frac{m_1 \bar{H} p}{B} = \frac{m_1}{B} \sqrt{\frac{8(p-1)M}{9}}$$

$$\text{trees: } T^* = \frac{m_1 \bar{H} p}{B} = \frac{m_1}{B} \left[ \frac{p R (p-2)^R}{(p-2)^R - 1} - \frac{p}{\ln(p-2)} \right]$$

$$\text{with } R = \frac{\ln[M(p-2)\ln(p-2)+p-1] - \ln(p-1)}{\ln(p-2)}$$

The equation for the first case again demonstrates that for any given  $M$ , a lower delay results if fewer ports are used. The equation for the second case however, requires a more detailed analysis.

Minimizing  $\bar{H}p$  by taking the derivative with respect to  $p$ , and solving this equation numerically yields the curve in figure 2.12. This curve gives the optimal number of ports (optimal in that it minimizes the average delay) as a function of  $M$ , the number of nodes communicated with.

#### TREE TOPOLOGY: OPTIMAL NUMBER OF PORTS

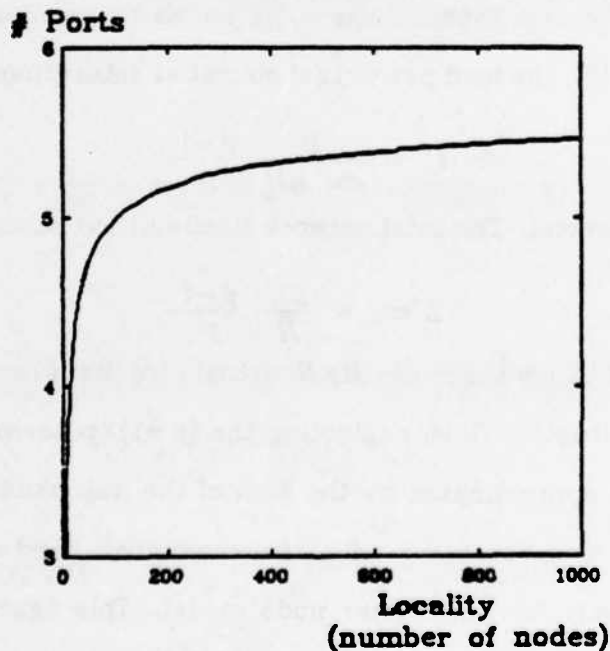


Figure 2.12. Optimal number of ports, tree topologies (no cut-through).

The above derivations assume that traffic from a node is uniformly distributed among the nodes it communicates with. In practice, one would try to map a specific problem onto the multicomputer in such a way that there is more traffic with nearby nodes than with those which are further away. Traffic between neighboring nodes should then be weighted more heavily. If one takes this into account, the case for the use of a few high-bandwidth links rather than many slower links becomes even stronger.

Thus, based on these studies, it appears that a communication component with relatively few ports, say from 4 to 6, is the most desirable. If cut-through is considered, the argument for a small number of ports also becomes stronger.

### 2.2.3.3. Bandwidth

Let us consider increasing the load on all virtual circuits of the network. As before, network bandwidth is defined as the asymptotic traffic load supported by the network as it approaches saturation, i.e. as link utilization  $\rho$  approaches 1. From equation (5), the load per virtual circuit at saturation  $l^*$  is

$$l^* = \frac{B}{m_i M \bar{H}} \frac{p-1}{p}$$

messages per second. The total network bandwidth at saturation is

$$L^* m_i = \frac{BN}{H} \frac{p-1}{p}$$

bits per second, since there are  $M \times N$  virtual circuits, where  $N$  is the number of nodes in the network. Thus, neglecting the  $(p-1)/p$  term, the total bandwidth of a network is approximated by the sum of the link bandwidths divided by the average hop count, agreeing with the maximum bandwidth figure of merit derived intuitively for the cluster node model. This figure is indicative of the maximum number of active virtual circuits the network can support at one time.

When comparing networks with the same number of chips, the sum of the link bandwidths is constant, so the topology with the smallest average hop count

will achieve the highest bandwidth. Thus, in this case, networks constructed with the largest number of links per node yield the most bandwidth. Bandwidths for tree and lattice networks are shown in figure 2.13 for various values of  $p$ , confirming this intuitive result. The curves also demonstrate how rapidly network bandwidth diminishes as traffic becomes less localized.

#### 2.2.4. M/G/1 Queueing Models

The queueing models presented thus far have assumed that message lengths are exponentially distributed. This allows one to use M/M/1 queueing models which can be easily solved. Since the networks described here use fixed length packets, an M/G/1 model is more appropriate. Unfortunately, the exact solution of complex networks of M/G/1 queues is unknown, since Jackson's theorem can no longer be applied. Briefly, Jackson's theorem allows one to solve a network of queues with Markov arrival rates by examining each queue independently, isolated from the rest of the network. Fixed length packets imply non-exponential service times which leads to non-Markovian behavior.

An alternative approach to resolving this dilemma is to use M/G/1 queues, but to make the approximating assumption that Jackson's theorem can still be applied. Other studies have indicated good correspondence between this model and simulation results [Klei78]. The Pollaczek-Khinchin mean value formula indicates that replacing an M/M/1 queue with one using fixed service times reduces the waiting time (i.e. the time spent waiting for the link to become free) by a factor of two [Klei75]. In the analysis presented thus far, this implies that equation (2) for the cluster node model becomes

$$T_i = \frac{1}{2} \left[ \frac{m_i}{C(1-\rho_i)} + \frac{m_i}{C} \right] - (1-\rho_{i+1})(t_m - t_h)$$

while equation (4) for the second model becomes

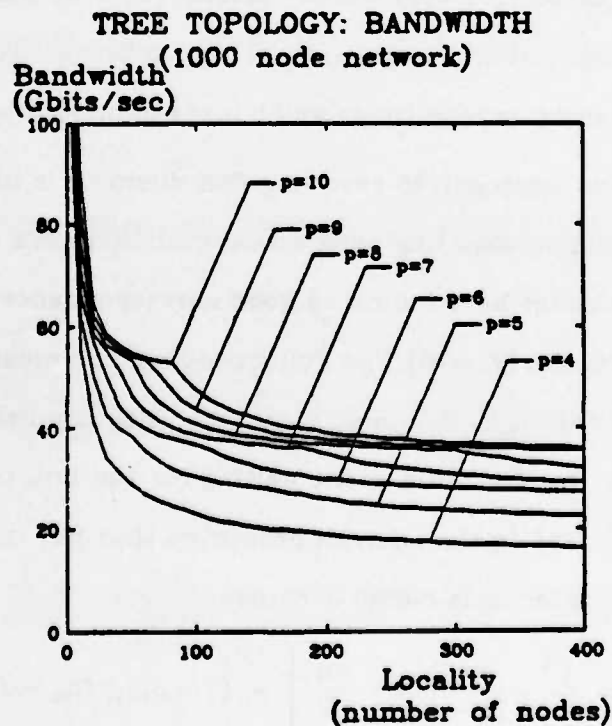
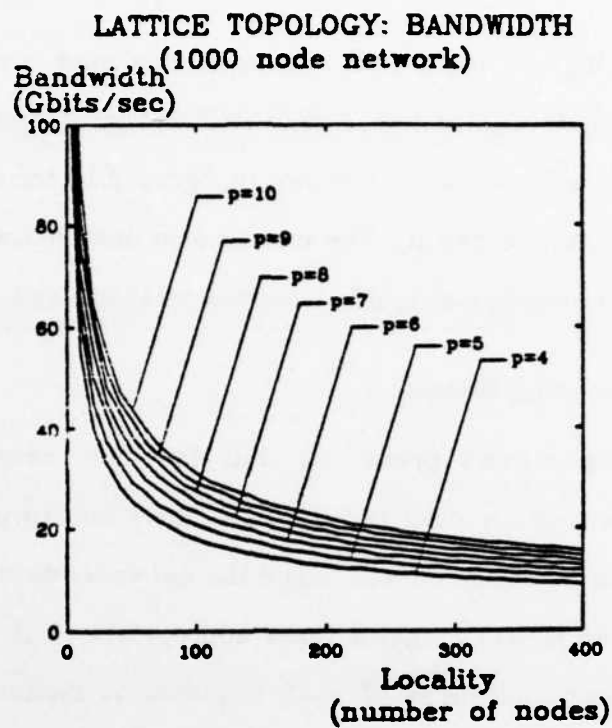


Figure 2.13. Bandwidth (#chips/hop count) (a) lattices (b) trees.

$$T = \frac{\bar{H}}{2} \left[ \frac{m_1 \rho}{B(1-\rho)} + \frac{m_1 p}{B} \right] - (\bar{H} - 1)(1-\rho)(t_m - t_h).$$

Closer examination of these equations reveals however, that delay in lightly loaded networks (delay as  $\rho$  approaches 0) and network bandwidth (traffic load as  $\rho$  approaches 1) are identical to that in the M/M/1 model. Thus, the M/G/1 queueing models yield curves with the same relative orderings as those derived for the M/M/1 models.

### 2.2.5. Summary of Analytic Results

The analytic results for the optimal number of ports are summarized in table 2.2 below.

**Table 2.2.**  
**Optimal Number of Ports**

model	delay	bandwidth
cluster nodes	small	small
fixed number of components	small	large

When considering delay, all of the analytical models presented here indicate that better performance is achieved with communication components with a relatively small number of ports, say from 3 to 6. Virtual cut-through reduces the impact of larger hop counts, and thus pushes the optimal number of ports closer to 3. It is seen that a cut-through mechanism can substantially reduce transmission delays in the network, so it is unreasonable to exclude it from any communication component design.

When considering bandwidth, the cluster node model favors components with the minimum number of ports, while the "one communication component per processor" model favors a large number of links. It is important to realize however, that the overall bandwidth of a network can be increased by adding

more chips, since the sum of the link bandwidths grows faster than average hop count in most topologies (rings, which are generally considered to be unsuitable for networks with a large numbers of processors, are an exception). This is verified by the cluster node model, where networks constructed from components with a small number of ports achieved greater bandwidth. Thus, achieving low latency appears to be the more important problem, leading to further support of communication components with a small number of ports.

It is important to remember that these bandwidth studies measure maximum network bandwidth, and thus only consider performance under heavy traffic loads. In a lightly loaded network, the bandwidth available to individual virtual circuits is equal to the bandwidth of the communication links it uses and thus will be larger if components with a small number of ports are used. Therefore, when combined with the analytic results presented in this section, one must conclude that providing general purpose communication components with a small number of ports, say from 3 to 5, is the best choice.

The analysis presented above made a number of simplifying assumptions. The strongest assumption concerned the traffic distributions among processors. Simulation studies which explore a number of different traffic distributions will be discussed next. It will be seen that for the most part, these simulations support the conclusions derived analytically. When discrepancies do occur, the simulations indicate better performance for components with a small number of ports, thus strengthening the conclusion that a small number of ports should be used.

## CHAPTER THREE

### SIMULATION STUDIES

The analytical models presented earlier made some simplifying assumptions. In particular, traffic distributions were assumed to be such that links are equally utilized, message arrivals were assumed to follow a Poisson distribution, and message lengths were assumed to follow an exponential distribution. To evaluate the conclusions derived by the analytical models when these assumptions are relaxed, and to gain deeper insight into the tradeoffs between various network topologies and realizations of the communication components, a simulation program was developed. The results of these simulation studies are discussed in this section. An instruction level simulator called Simon is described, and the respective speedups resulting from executing several parallel application programs on various network structures are reported.

The first two sections describe the simulator and the assumptions made about the multicomputer system. Following this, the application programs are described, and simulation results are presented. Some of the issues evaluated by this study include the optimal number of ports and the effect of incorporating a mechanism in the communication hardware for efficiently handling multiple-destination messages.

#### 3.1. The Simulator: Simon

Simon (Simulator of Multicomputer Networks) is a discrete-time, event-driven simulation program designed to facilitate comparison of alternate switching structures [Fuji83]. The most important features of Simon are:



- (1) Traffic in the communication domain is generated by application programs executing some parallel algorithm. This is in contrast to the analytical studies which made the simplifying assumptions that links are equally loaded and message arrivals follow a Poisson distribution.
- (2) The software modeling the interconnection network is contained in a separate module called the "switch model", allowing easy comparison of different switching structures.

The simulator consists of three components (see figure 3.1): the application program, the simulator base, and the switch model. The application program consists of a number of tasks, or equivalently, processes, which execute in parallel and communicate by exchanging messages. The simulator base time-multiplexes execution of the tasks on the host computer, in this case a VAX-11/780. The base also keeps track of time for each task (each task has a clock which advances as the task executes) to ensure that interactions among tasks (e.g. message transmissions) are simulated in the proper time sequence. Finally, the switch model provides a fixed virtual circuit interface for the tasks and simulates message passing between processors. A detailed description of the simulator is given in [Fuji83].

### 3.2. Assumptions

A number of assumptions are made in the simulation experiments reported here. These include:

- (1) negligible operating system overhead
- (2) VAX 11/780 processing elements
- (3) one-to-one mapping of tasks to processors
- (4) fixed length packets (1 byte header, 16 data bytes)

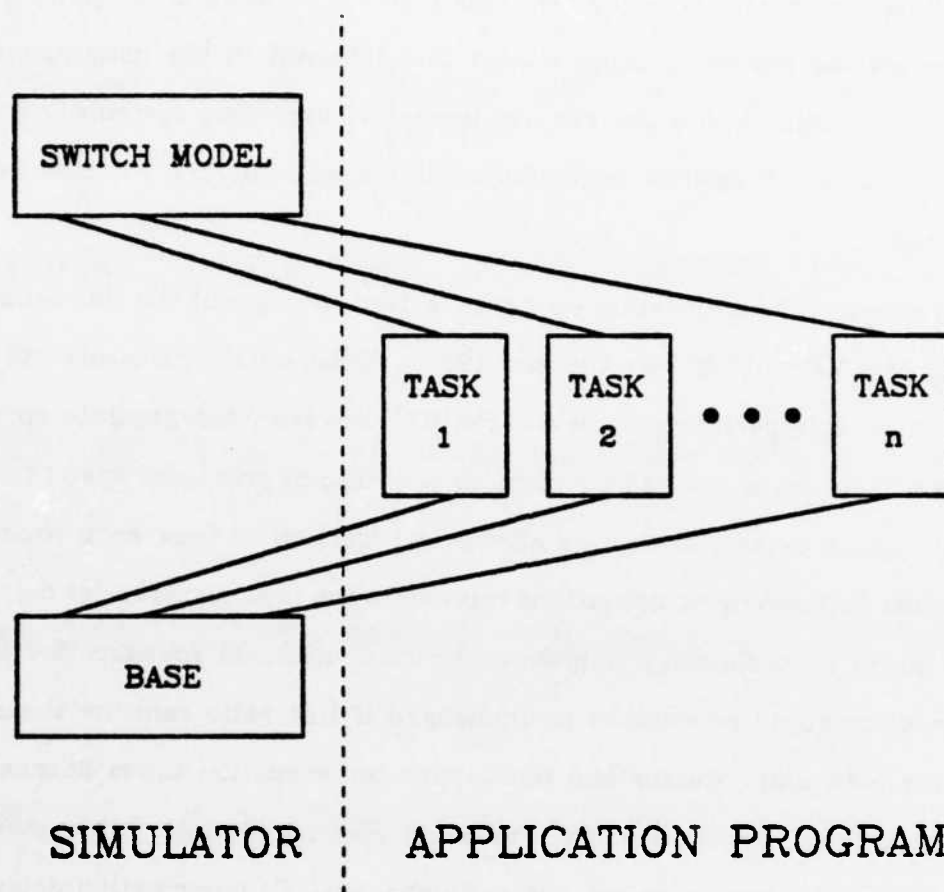


Figure 3.1. *Block diagram of simulator.*

- (5) unlimited buffering within each communication component
- (6) error free transmission
- (7) virtual cut-through
- (8) virtual circuits set up in advance
- (9) shortest path routing

Each of these assumptions will now be discussed in turn.

- (1) The bulk of the simulation studies assume that the time to execute an operating system routine for invoking a communication mechanism (e.g.

sending a message) is negligible. This allows separation of the penalty due to operating system overhead from that inherent in the communication switch. Studies which analyze the impact of operating systems overhead alone, i.e. which assume negligible communication delays, will also be discussed.

- (2) The speed of the processing elements is fixed throughout the simulations to that of a VAX-11/780. By the mid 1980's, 32-bit microprocessors will have achieved this performance level [Patt82]. However, the absolute speed of the processors is not so important as the ratio of processor speed to communication delays, since this affects the fraction of time each processor spends performing computations relative to the time required for communications. As technology improves, the computational speedup due to the use of multiple processors is unchanged if this ratio remains the same, since both uniprocessor and multicomputer execution times decrease by the same factor. If however, processor speed increases at a faster rate than communication speed, the ratio changes. Communication delays will prevent the multicomputer execution time from decreasing in proportion to that of the uniprocessor, and speedup actually decreases. Here, since the "VAX" assumption implies a constant processor speed, this ratio is changed by varying communication bandwidth. This provides the flexibility of determining performance under 1983 technology, as well as predicting the effect of technological changes.
- (3) It is assumed that each task executes on a separate processor. In other words, it is assumed that the system contains enough processors to accommodate the application program. The programs studied here use at most 32 processors, so this is a reasonable assumption. Indeed, general purpose systems using more than 32 processors have already been constructed

[Swan77a, Stri83, Kush82, Hosh83].

- (4) Packets consist of a single control byte followed by 16 data bytes. Fixed-size packets are used because of the difficulties associated with managing variable sized buffers, as discussed in chapter 4. This is in contrast to the analytic models described in chapter 2 which made the simplifying assumption that message lengths follow an exponential distribution. The control byte is used to specify a virtual channel number, as will be discussed in chapter 4. In the application programs discussed here, messages are short, typically consisting of only a single floating point number, and fit within a single packet. An area of future research is to consider workloads which include large, multi-packet messages, e.g. paging traffic and/or file transfers.
- (5) It is assumed that adequate buffer space is available in each component for holding packets waiting to be forwarded. It will be shown later that chip densities now allow each component to provide enough buffer space to achieve approximately the same performance as a component with an unlimited amount of buffering.
- (6) The simulator assumes that no errors occur during data transmissions. This assumption was also used in the analytical models, and was justified in the discussion there.
- (7) Virtual cut-through is used in all networks. Partial cut-throughs are allowed, i.e. if the outgoing link used by a packet is busy when the header arrives, but becomes free before the tail arrives, the packet need not wait for the latter event before it begins using the link. The analytical results presented earlier indicated that substantial improvements can be achieved with virtual cut-through, so it is unreasonable to exclude it from any design.

- (8) It is assumed that all virtual circuits are set up before the tasks begin execution. All of the application programs studied are static in the sense that new tasks are not created after execution begins. Since the programs execute for long periods of time, the set-up time is negligible relative to the total execution time. Thus, its effect on overall performance can be neglected.
- (9) Finally, the simulator uses a shortest path routing algorithm to set up its virtual circuits. Within the simulator, Floyd's algorithm [Floy62] is used to perform this computation. To prevent unfair comparisons, one routing algorithm was used throughout all of the simulation studies. Shortest path routing was selected because it has a simple implementation and because it has some prospect of achieving good performance since it minimizes the amount of network resources, i.e. bandwidth, required for each virtual circuit. Evaluation of more sophisticated routing algorithms is a topic of future research.

### 3.3. The Application Programs

Traffic distributions are generated by application programs executing parallel algorithms. For the purposes of this study, an application program is characterized by the communication pattern it generates. In particular, communications are characterized by the structure of communications between the program and its surrounding environment, and the pattern of communications within the program, i.e. among its tasks.

External communications between the parallel program and its environment are assumed to fall into one of two categories:

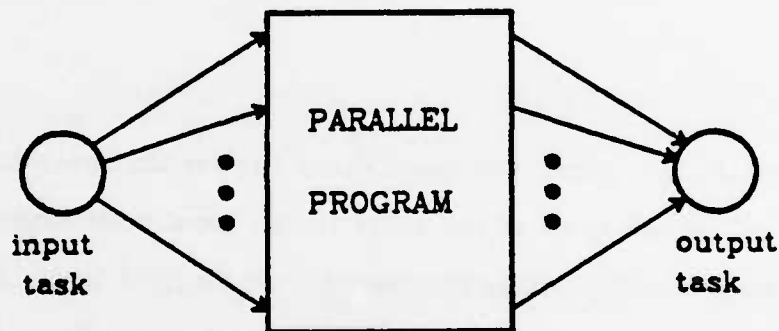
- (1) serial input, serial output (SISO).

(2) parallel input, parallel output (PIPO).

These two communication patterns are shown in figure 3.2. In SISO, the input data arrives from (is sent to) a single source (destination). In PIPO, the data arrives (leaves) in parallel from (to) several sources (destinations).

Several of the application programs implement signal processing functions which use an SISO communication pattern. A single processor samples the input waveform and distributes the data values to a number of the other processors

### SERIAL INPUT SERIAL OUTPUT (SISO)



### PARALLEL INPUT PARALLEL OUTPUT (PIPO)

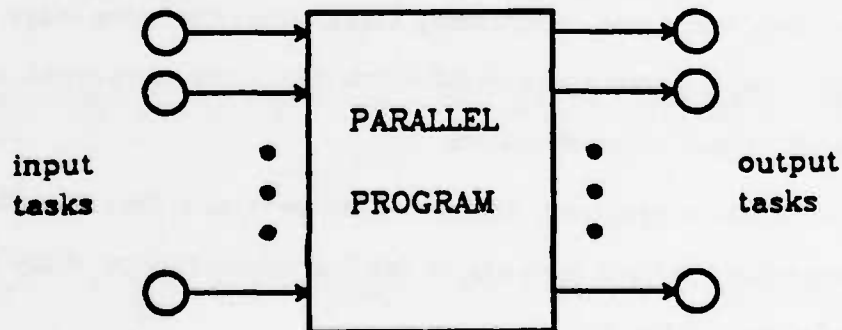


Figure 3.2. Communication patterns for application programs.

which collectively compute results. Another processor collects the output waveform. In other situations, a PIPO structure might arise. For example, the application program could be one of several job steps, each of which is implemented as a separate parallel program. Since the input (output) of each job step comes from (goes to) another parallel program, one can expect data to arrive (leave) in parallel. While other communication patterns are possible, e.g. SIPO or PISO, these are only combinations of the patterns presented above, and are not fundamentally different.

The internal communication paths are also partitioned into two categories:

- (1) global
- (2) local

As the name implies, global communications implies that each task communicates with all, or nearly all of the other tasks. Local communications implies each task communicates with a small subset of the other tasks. The programs studied here that use local communications are pipelined. Thus, the communications are local in the sense that each stage of the pipeline sends messages only to the next stage, and not to previous or subsequent stages. Although this communication structure does not exhibit loops among tasks in different stages of the pipeline, loops may exist among tasks within the same stage. Programs that exhibit loops among tasks in different stages are considered to belong to the class with global communications.

Six application programs demonstrating several different traffic patterns were run on Simon. Each uses one of the four combinations of the parameters described above. These are:

- (1) Barnwell, a signal processing program using Barnwell's algorithm (global SISO)



- (2) Block I/O, a signal processing program using block filters (local SISO)
- (3) Block State, a second program also using block filters (local SISO)
- (4) FFT, a program for computing Fast Fourier Transforms (local PIPO)
- (5) LU, a program for performing LU decomposition on a sparse matrix (global PIPO)
- (6) Random, a program generating artificial traffic loads (global PIPO)

The communication patterns exhibited by these programs are summarized in table 3.1 below.

**Table 3.1**  
**Communication Structures**  
**Used by the Test Programs**

	SISO	PIPO
<b>global</b>	Barnwell (12 tasks)	Random (12 tasks) LU (15 tasks)
<b>local</b>	Block I/O (23 tasks) Block State (20 tasks)	FFT (32 tasks)

All of these programs communicate relatively small amounts of data frequently. Typically, a task waits for data values to arrive from other task(s), performs some floating point operations on them, and then generates a result which is passed on to another task(s). The number of processors ranges from 12 in the Barnwell program to 32 in the FFT. Each of these programs will now be discussed in greater detail.

### 3.3.1. Barnwell Filter Program (global SISO, 12 tasks)

The Barnwell filter, and the two programs which follow, implement the digital filter defined by the equation:

$$Y_n = \sum_{i=1}^{N-1} b_i Y_{n-i} + \sum_{i=0}^{M-1} a_i X_{n-i}$$

Vectors  $X$  and  $Y$  are the input and output waveforms,  $A$  and  $B$  characterize the

filter being implemented, and  $N$  and  $M$  are the number of poles and zeros in the filter respectively. The programs presented here use  $M = N = 7$ .

An "input task" distributes a total of 400 samples of the input waveform (the real multicomputer would collect this data from a sensor at some sampling frequency) to some number of "computation tasks". An "output task" collects the output waveform computed by the computation tasks. Thus, all three programs have an SISO communication pattern. When all of the 400 input samples have been processed, execution terminates. It is assumed that the sampling frequency is large compared to the rate at which data points can be processed. This ensures that the execution time is not limited by the input data rate. Thus, at time 0, the input processor begins distributing the 400 data points to the computation processors and never waits for input data.

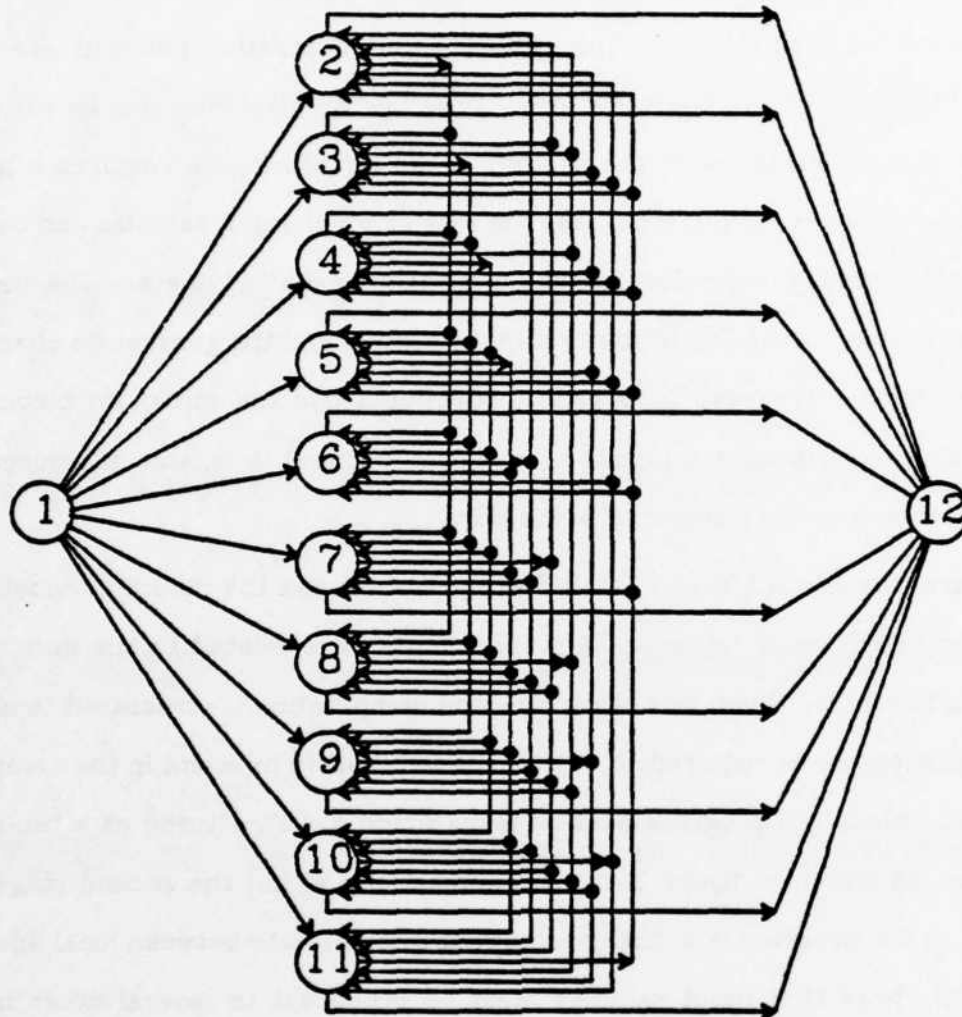
The Barnwell program computes the filtering function using Barnwell's algorithm [Barn78, Barn79, Hodg80, Barn82, Lu83]. The two signal processing programs which follow use a different technique for performing the calculations. In Barnwell, twelve tasks are used, as shown in figure 3.3. Each node in figure 3.3 represents a task, and each arc a virtual circuit. An arc which fans out to several destinations represents a broadcast communication.

The Barnwell program uses ten tasks to execute the signal processing algorithm. This is the maximum number of processors the algorithm can effectively use in performing the computation, assuming small communication delays. This number is a function of the number of poles in the filter being implemented. Each computation processor receives 40 input samples.

Each data point received by a computation processor is combined with data generated by other processors. The result is then broadcast to the six processors immediately "to the right" of that processor. These communication paths are shown in figure 3.3. The communication pattern for the Barnwell program is

# BARNWELL PROGRAM

89



**Figure 3.3.** *Communication paths for Barnwell program.*

classified as global SISO, although communications are really only approximately global since each computation processor does not communicate with all others.

## 3.3.2. Block I/O Filter Program (local SISO, 23 tasks)

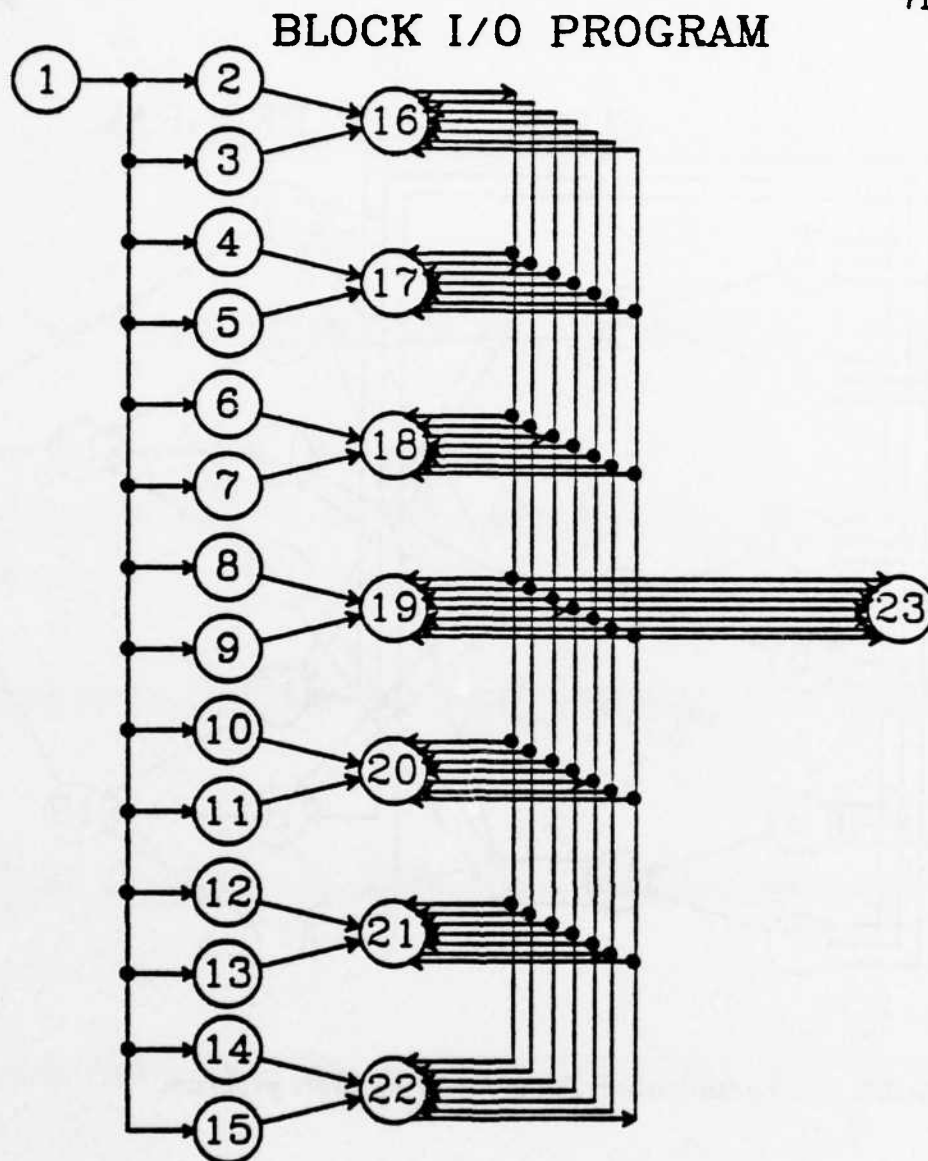
The Block State and Block I/O programs perform the filtering function described above by grouping the input samples into blocks and then processing

each block as a single unit. The resulting communication patterns are local SISO. These algorithms have the advantage that the block size can be varied to change the performance of the system. A larger block size requires a larger number of processors, but increases the rate at which input samples can be processed. Increasing block size does incur a latency penalty however. The amount of time between reception of the first input sample and the generation of an output waveform increases. In practice, one would use the minimum block size which allows the input samples to be processed in real time; this minimizes the latency as well as the number of processors.

Here, the Block I/O and Block State programs use the minimum block size in order to minimize latency. This minimum size is related to the number of poles in the filter. Given this block size, the computation is structured to use as many processors as required to exploit the parallelism inherent in the computation. The Block I/O program uses 23 tasks which are structured as a two-stage pipeline, as shown in figure 3.4. Communications within the second stage are global, so the program is actually somewhat intermediate between local and global SISO. Note that input samples must be broadcast to several other tasks. Details of the algorithms implemented by this program can be found in [Burr71, Burr72, Mitr78, Lu83].

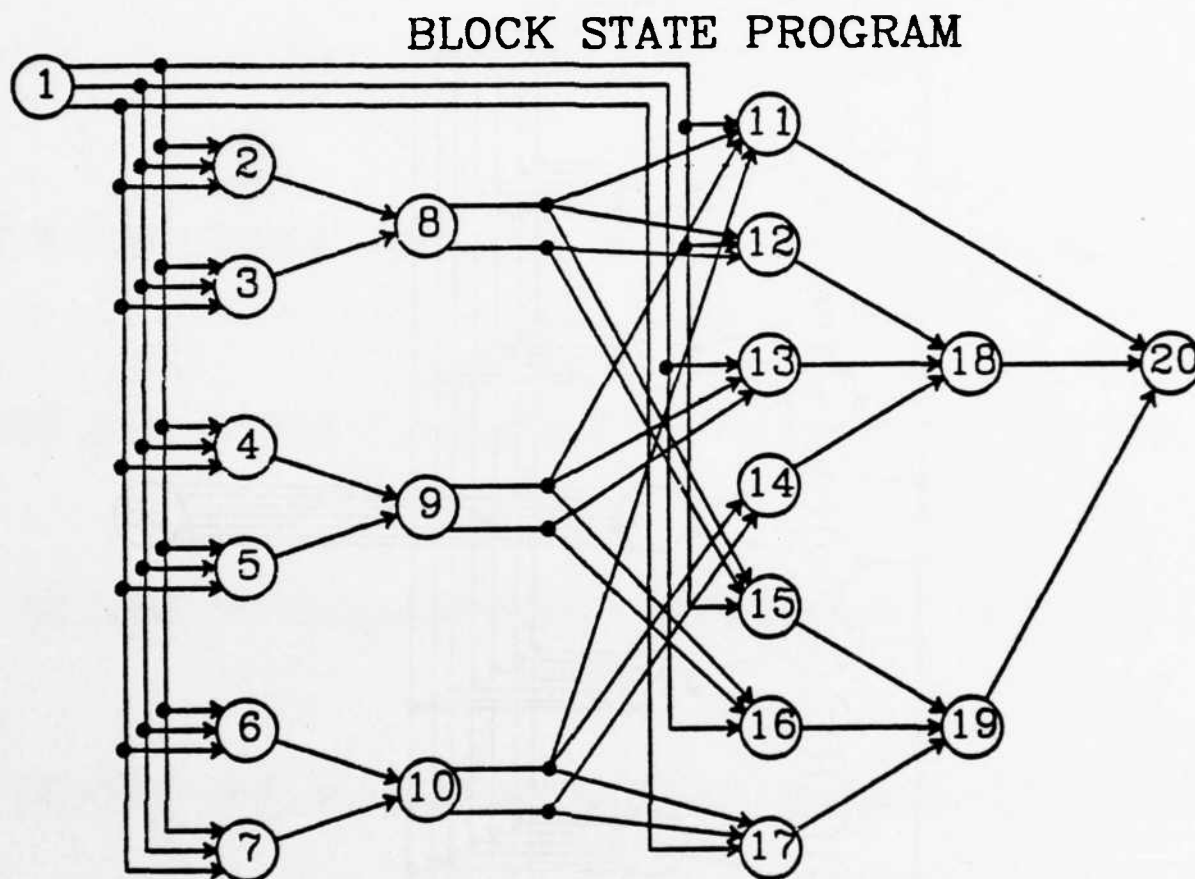
### 3.3.3. Block State Filter Program (local SISO, 20 tasks)

The Block State program uses the same "blocking" techniques discussed in Block I/O. This program however, uses a somewhat different approach to perform the computation, and as a result includes information of the internal behavior of the filter as well as the input-output relationships. Thus, it allows the determination of some intermediate values which the Block I/O program does not compute. As before, the minimum block size is used, resulting in a computation which requires 20 tasks. The communication paths for this



**Figure 3.4.** *Communication paths for Block I/O program.*

program are shown in figure 3.5. It is seen that the computation uses a 4 stage pipeline, and thus exhibits a local SISO communication pattern. Again, input samples are distributed via multiple-destination messages. Further details of the algorithms used in the Block State program can be found in [Barn80a, Barn80b, Zema81, Lu83].



**Figure 3.5.** *Communication Paths for Block State program.*

### 3.3.4. FFT Program (local PIP0, 32 tasks)

This program performs a complex 16 point Fast Fourier Transforms on sets of input values. The FFT algorithm is used to compute the Fourier coefficients for an analog signal. The input consists of 400 sets of complex input values,  $x_0 \cdots x_{15}$ . The output consists of 400 sets of complex numbers  $y_0 \cdots y_{15}$ , such that

$$y_i = \sum_{k=0}^{15} x_k \exp((-2\pi/16)ik)$$

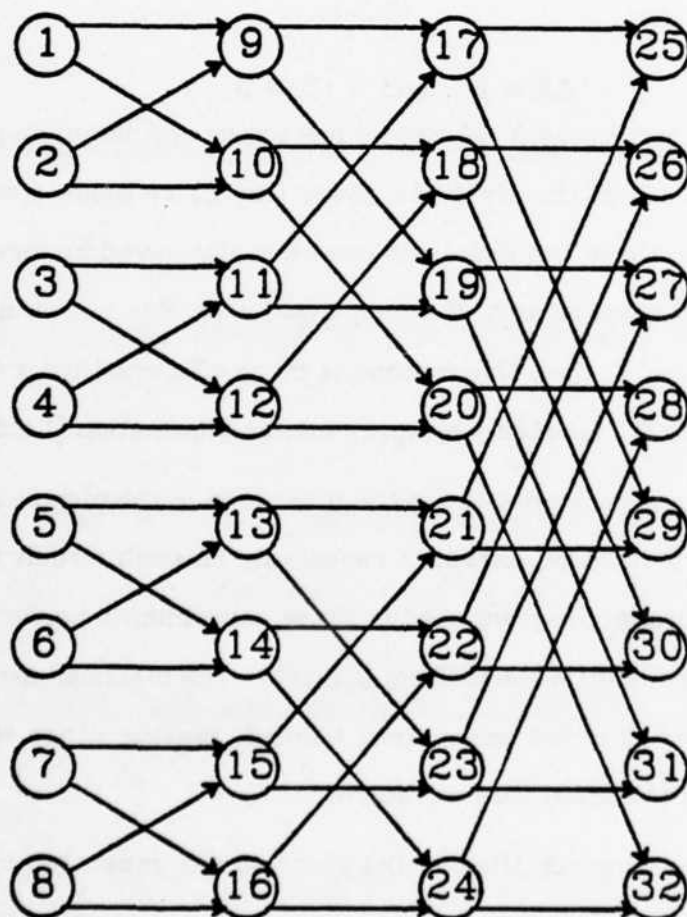
Details of the algorithm used to perform this computation in time proportional



to  $N \log N$  (here,  $N=16$ ) are discussed in e.g. [Baas78].

The communication paths used by this program are shown in figure 3.6. Since the same computation is performed on several sets of input data, the computation can be pipelined. The input data are assumed to reside in the processors comprising the first stage of the pipeline, so the resulting communication paths are local PIP0.

### FFT PROGRAM



**Figure 3.6.** *Communication paths for FFT program.*



### 3.3.5. LU Decomposition (global PIPO, 15 tasks)

This program performs LU decomposition on a sparse matrix. LU decomposition is a well known technique for solving a set of linear equations. Suppose a set of equations is specified as

$$AX = Y$$

where  $A$  is a known  $n$  by  $n$  matrix,  $Y$  is a known column vector of length  $n$ , and  $X$  is an unknown column vector also of length  $n$ . The solution to this equation can be found by factoring the  $A$  matrix into two components,  $L$  and  $U$ , and then solving the equations

$$LB = Y \quad \text{and} \quad UX = B$$

in turn for  $B$  and then for  $X$ .  $L$  and  $U$  are upper and lower diagonal matrices respectively, i.e. all of the elements above (for  $L$ ) or below (for  $U$ ) the main diagonal are 0, so these two equations can be easily solved by forward and backward substitution respectively. If the equation  $AX = Y$  is solved many times with different values for  $Y$ , then this method is more efficient than solving the original equation ( $AX=Y$ ) repeatedly by say, gaussian elimination [Dahl74].

LU decomposition is one step in the inner loop of the circuit simulation program SPICE, so it must be executed repeatedly on each circuit simulation run [Nage75]. The parallel program used in these experiments performs the decomposition by using Doolittle's algorithm [Chua75]. The matrices used in this application are sparse, but not necessarily banded, making other techniques, e.g. systolic methods [Mead80], less attractive.

Given a sparse matrix, the parallel program was generated by first creating uniprocessor code for performing the computation, analyzing the data dependencies within this code, and then creating a parallel program from the data dependency graph [Yu84, Wing80]. For the program in question, the communication pattern which results from this process is global, i.e. every task sends mes-

sages to every other task. The program is PIP0 since LU decomposition is only one of several parallel job steps in the inner loop for SPICE. Input (output) values can be expected to arrive from (be sent to) another parallel program executing the previous (subsequent) step of the inner loop.

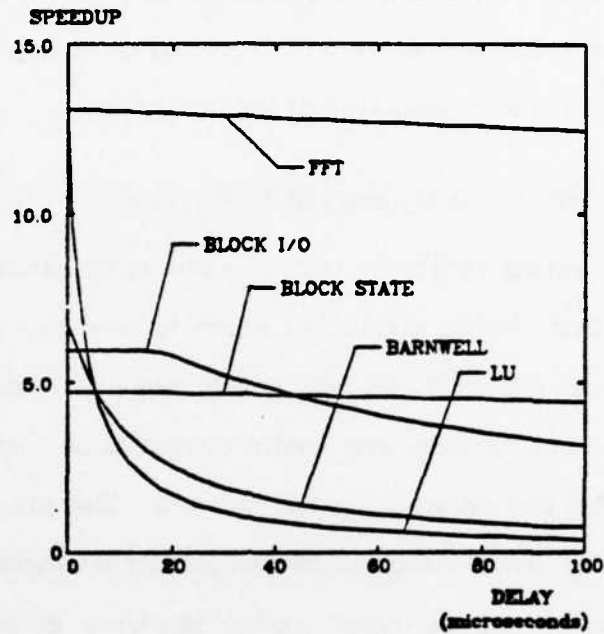
### **3.3.6. Artificial Traffic Loads (global PIP0, 12 tasks)**

A program creating synthetic traffic loads using random number generators was also studied. In the discussion which follows, this program is referred to as the "Random" program. In contrast to the other application programs, this program does not perform any useful computation. Its only function is to generate traffic for the communication network. The program consists of 12 tasks, each of which sends a total of 500 single-packet messages. Messages are uniformly distributed among other tasks, implying global communications. Since each processor originates its own messages, in contrast to a single processor generating all messages, the external I/O structure is PIP0. The mean time between messages is chosen from an exponential distribution. Loading on the network is increased by reducing the average time between messages.

### **3.4. Communication Delays**

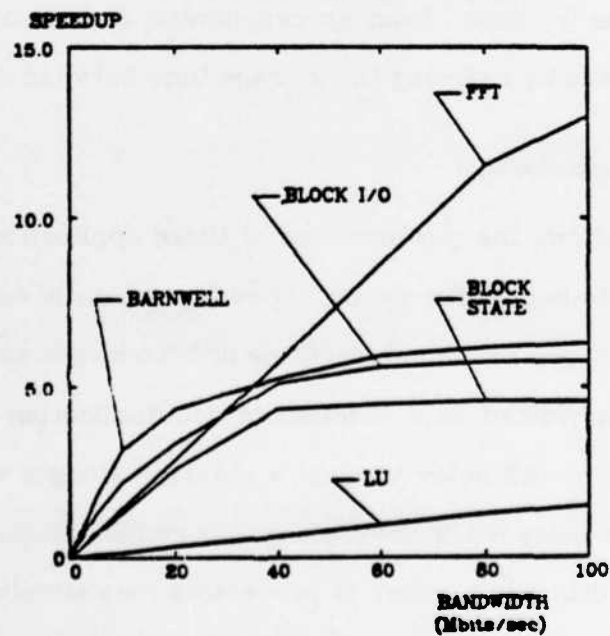
Figure 3.7a shows the performance of these application programs using a fixed-delay, infinite-bandwidth switch. Speedup, which is defined as the execution time of the program on a uniprocessor divided by the execution time on the multicomputer, is plotted as a function of communication delay. Here, delay refers to the end-to-end delay to send a message along a virtual circuit. It is assumed that this delay is the same along all circuits. Since the switch provides unlimited bandwidth, any number of processors may simultaneously send messages.

## SPEEDUP vs. DELAY



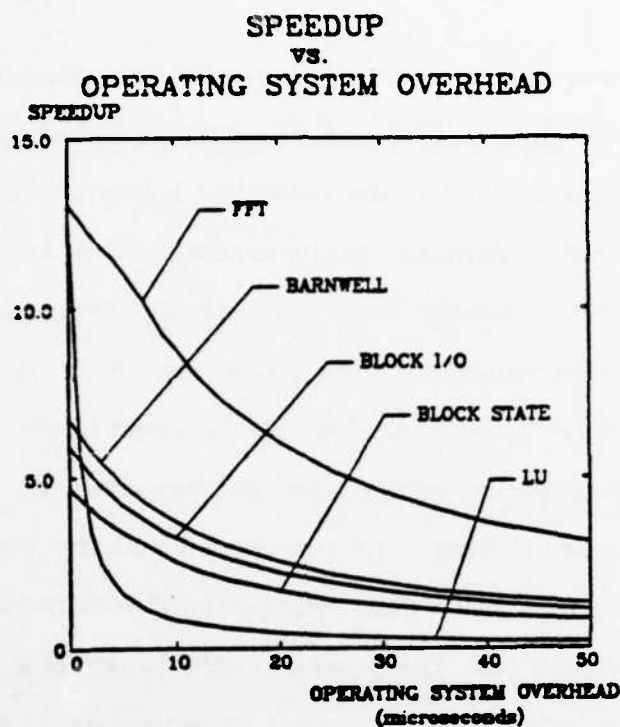
(a)

## SPEEDUP vs. BANDWIDTH



(b)

Figure 3.7. Speedup vs. (a) delay. (b) bandwidth.



(c)

Figure 3.7. (c) *Speedup vs. operating system overhead.*

The two programs using global communications, Barnwell and LU, experience a severe degradation in performance as communication delays increase. This results from the relatively fine "granularity" of the computation, in which communications are frequent and delays have a significant impact on total execution time. On the other hand, the FFT and Block State programs exhibit little performance degradation as delays increase. These programs are pipelined, so delays only affect the amount of time required to fill and empty the pipe. Once the pipeline is filled, data arrives at each processor at a constant rate, independent of communication delay, so all of the processors remain busy. It is erroneous however, to conclude that the interconnection switch does not impact the performance of these programs, since the curves in figure 3.7a assume unlimited network bandwidth.

The curves in figure 3.7b show the performance of the programs as a function of network bandwidth. Conceptually, the network can be viewed as an entity

which provides a certain amount of bandwidth (this quantity is plotted on the horizontal axis in figure 3.7b) for transmitting messages. The optimistic assumption is made that all of the network's bandwidth can be allocated to a single virtual circuit on demand. In the simulator, this is implemented by using an "ideal bus" switch model. The communication network consists of a single bus of the indicated bandwidth. The full bandwidth of the bus is allocated to messages as they are generated. Conflicts to access the bus are queued in FIFO sequence, and propagation delays along the bus are assumed to be zero. The curves indicate that although the performance of the pipelined programs is insensitive to communication delay, adequate network bandwidth is required to achieve good performance. The programs exhibiting global communication patterns behave similarly. The LU program in particular, is seen to require very large amounts of bandwidth before achieving good performance. Simulations at higher bandwidths indicate that a 500 Mbit/second network is required to achieve a speedup of 10.0 (speedup with an infinite bandwidth, zero delay switch is 12.7).

Finally, the curves in figure 3.7c indicate performance as a function of operating system overhead. Here, overhead is measured as the time required to execute an operating system routine for sending or receiving a message. Transmission delays are assumed to be zero. It is seen that degradation is severe when delays in the operating system are only a few tens of microseconds. This result again is a consequence of the relatively fine granularity of the computation. It points out that hardware support for operating system primitives (here sending and receiving messages) is required to allow full exploitation of the parallelism inherent in many programs. With a traditional software implementation, the time spent in the operating system will dominate the transmission time, negating the benefits of incorporating a high-performance communi-

cation network. In particular, since recovery from transmission errors is left to an end-to-end protocol, hardware support should be employed in the computation processor to keep these checks from degrading performance. Hardware support for communication primitives thus represents an important area of future research.

### 3.5. Issues Under Investigation

Four separate issues are studied in these simulation experiments. The first explores the optimal number of ports, and compares simulation results with those predicted by the analytical models presented earlier. Next, an alternative model in which processor and communications are integrated onto the same chip is studied. Third, since many of the application programs send the same message to several different destinations, the impact of incorporating a mechanism which efficiently handles such messages, i.e. a multicast mechanism, is investigated. Finally, the particular mapping of tasks to processors which was used in these experiments is examined, as well as its impact on the simulation results.

In order to evaluate the optimal number of ports, two types of switch models were implemented: cluster nodes and networks with a fixed number of components. These switch models correspond to the networks discussed in the analytical studies presented earlier. In the first, each node of a topology requiring  $b$  branches per node is implemented with a cluster of  $p$ -port communication components. As  $p$  is reduced, the number of components required to construct the network is increased. Thus, the cluster node switch models do not keep the chip count constant. The second set of switch models compares networks with different values of  $p$ , but with approximately the same number of components.

In addition to networks constructed from separate computation and communication components, networks with processor and communications

integrated onto the same chip are studied. This is the building block for the "network computer" proposed by Wittie [Witt81]. In this model, the communication links between the computation and communication domains are eliminated. In communication component networks, it will be seen that these links sometimes become bottlenecks which bias the results. The simulations under this latter model eliminate this bias. Multicomputers using the Wittie model do require more circuitry per chip than those using communication components, making direct comparisons unfair. Nevertheless, it is included as an alternative model for multicomputer networks.

Since the digital filtering algorithms (Barnwell, Block State, and Block I/O) involve transmitting the same data to several destinations, a mechanism which efficiently distributes multiple-destination packets (i.e. a "multicast" mechanism) is expected to improve performance.

If a multicast mechanism is *not* used, several "single destination" packets are generated at the source node, one for each destination, and each is routed separately through the network to its particular destination using a shortest path routing algorithm. If one traces the paths followed by these packets through the network, it is seen that packets will follow each other up to a certain point, at which time they part and go their separate ways. The multicast mechanism combines the single destination packets which are "following each other" into a single "multicast packet". A new copy is not generated until one or more of the single destination packets incorporated into the multicast packet need to "go their separate ways". If several packets breaking off like this are all going in the same direction, only one new multicast packet is created. Multicast and broadcast mechanisms are described more fully in [Dala78, Bhar83, McQu78]. Note that since virtual circuits are used, implementation of this does not affect other parameters of the switching network. A longer header might be



needed to provide a list of destination nodes. However, in a virtual circuit mechanism, this information need only be carried through the network when the multicast circuit is set up.

The mapping of the application program onto the network topology is identified by labels assigned to tasks and processors. As shown in figures 3.3-3.6 (the remaining two programs, Random and LU, use global communications, so the mapping does not influence the results), each task of each application program is characterized by a unique integer called its "task id". Similarly, each node, i.e. processor, of a topology is characterized by a unique node number. In the discussions which follow, task  $i$  always executes on processor  $i$ . Thus, the simulation results assume a specific mapping of tasks to processors. Care must be taken to ensure that this mapping does not bias the results. More will be said about this later.

### 3.6. Simulation Results on Cluster Node Networks

As discussed earlier, one can implement a node of a topology requiring  $b$  branches per node as a cluster of  $p$ -port communication components. The various application programs described above were run on Simon using switch models for several different cluster node networks. The results of these simulation experiments are reported in this section.

For this study, four topologies are examined which vary  $b$  over a wide range of values. All topologies are assumed to use full duplex, bidirectional links. These topologies are:

- (1) Fully connected network.
- (2) Full-ring binary tree [Desp78].
- (3) Butterfly network.

#### (4) Ring network.

The topology within each cluster node is a balanced tree, with the processor attached to the communication component at the root.

In all of the graphs which follow, speedup is plotted as a function of  $B$ , the total I/O bandwidth of the communication chip. The only exception is the artificial traffic load program in which average message delay is plotted as a function of traffic load. It is assumed that the bandwidth  $B$  is equally divided among the existing communication links. Thus, a Y-component with  $B$  equal to 300 Mbit/second has three 100 Mbit/second communication links. For comparison, the speedup on a multicomputer with an infinite-bandwidth, zero-delay interconnection system (i.e. a "perfect switch") is also shown. The perfect switch assumes that messages arrive at their destination at the instant at which they are sent. It thus gives an upper bound on performance for any communication network.

The analytical results presented earlier indicated that cluster node networks constructed from communication components with a small number of ports yielded the most bandwidth and least delay. Thus, one would expect networks constructed from Y-components to yield the best performance. It will be seen that the simulation results confirm this conclusion.

##### 3.6.1. Fully Connected Networks

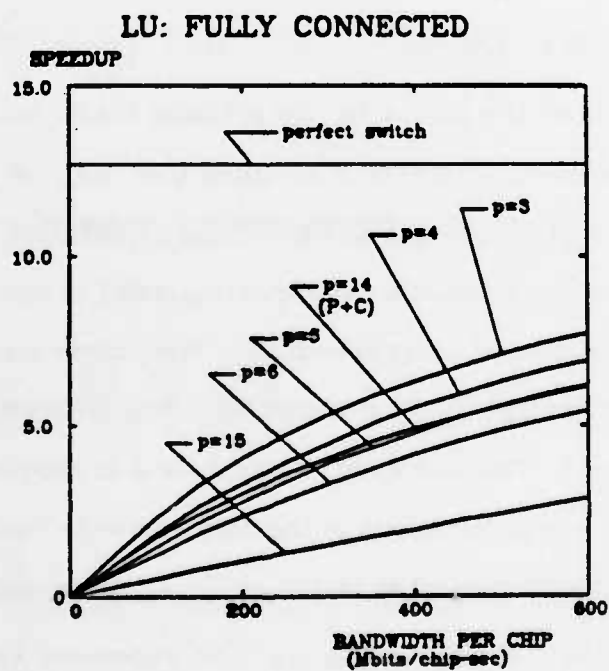
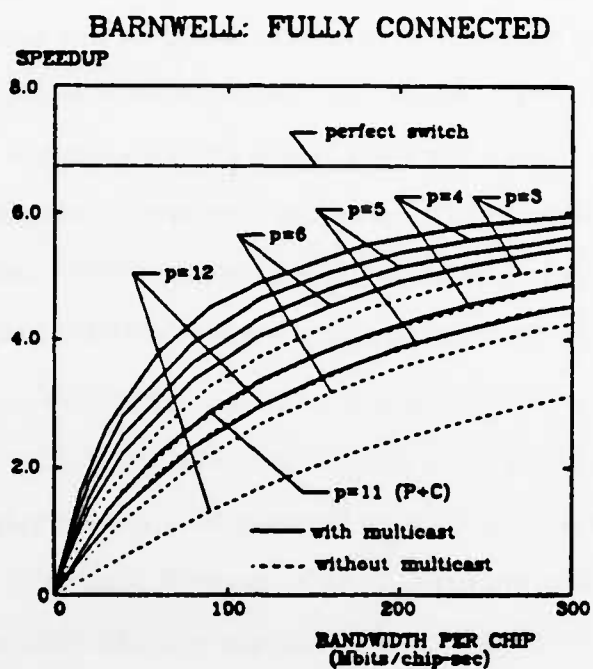
The fully connected network is formed by placing a single link between every pair of nodes. Here, the number of nodes is equal to the number of tasks required by the parallel program, and it thus varies from application to application. This topology minimizes the number of hops between every pair of nodes, but at the expense of a larger number of branches on each node.

Three of the application programs were run on Simon with switch models for fully connected networks. Performance curves are shown in figures 3.8a-c. Due to limited amounts of computing resources, cluster node simulations for the FFT, Block I/O, and Block State programs are not available. Figures 3.8a-c indicate that performance improves as the number of ports is reduced in agreement with the analytical results. Curves labelled "P+C" indicate that processor and communications circuitry are incorporated onto the same chip.

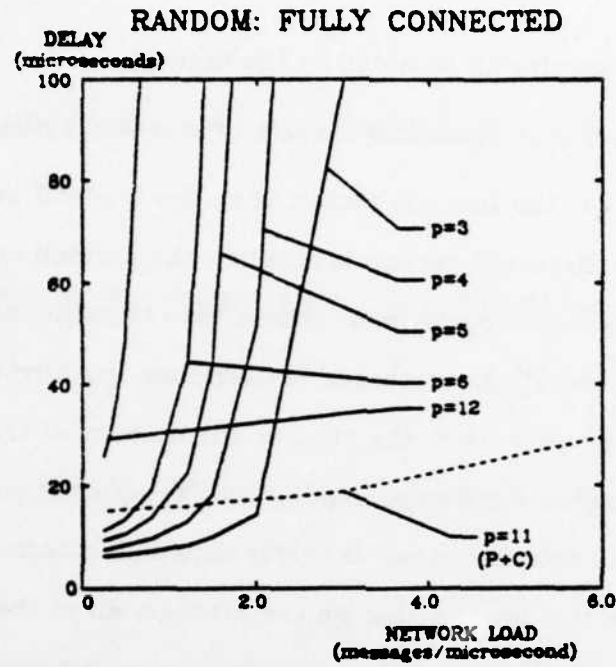
The curves resulting from the artificial traffic load program indicate that reducing the number of ports reduces the average delay in a lightly loaded network, and increases total network bandwidth. The bandwidth result however, is somewhat misleading because the total network bandwidth shown in figure 3.8c is limited by the link between the processor and its communication component. This is demonstrated by the curve in which communication circuitry is included in the same chip as the processor. Network bandwidth is increased significantly when this bottleneck is removed.

Figure 3.8d shows the curves for the artificial traffic load program with this bottleneck link removed. Here, it is assumed that the root component of each cluster node has both computing and switching capabilities. Other components only perform switching functions. As expected, delay in lightly loaded networks improves as the number of ports is reduced. The curves also indicate, however, that networks with a large number of ports provide as much bandwidth as those using Y-components. This unexpected result is a consequence of the lack of store-and-forward communications in the fully connected network, and the particular traffic distribution created by the artificial traffic load generator.

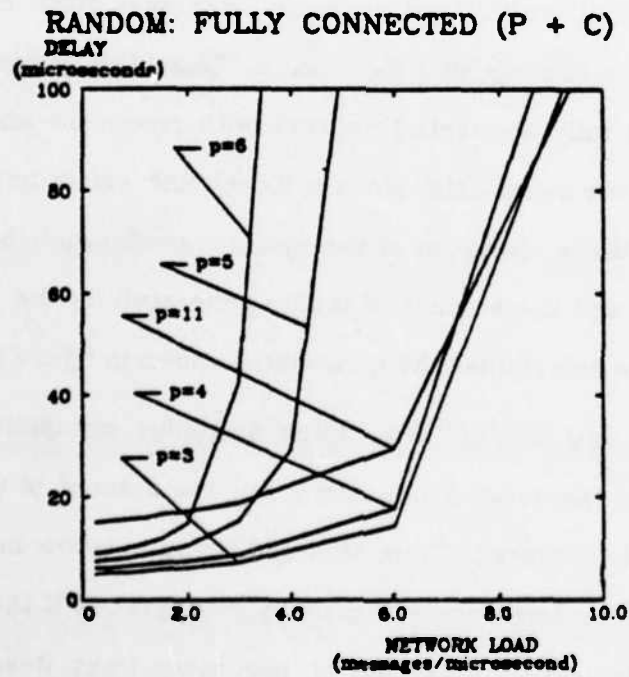
The bandwidths indicated by figure 3.8d represent the minimum of two quantities:



**Figure 3.8.** Fully connected network (a) Barnwell. (b) LU.



(c)



(d)

**Figure 3.8.** Fully connected network (c) Random. (d) Processor with Communications.

- (1) The maximum bandwidth provided by the network.
- (2) The maximum rate at which traffic can be generated by the processors.

If the first quantity is the limiting factor, then the tradeoff between hop count and link bandwidth discussed earlier determines the optimal number of ports. If the second quantity limits performance, then the utilization of the links around the processors generating messages determines performance. The more efficiently these links are used, the greater the amount of traffic sent into the network, and the higher the overall bandwidth. This quantity is maximized if the traffic generated by each processor is evenly distributed across that processor's output links, since this implies that on the average, all of the processor's links will be busy all of the time. An uneven distribution causes some links to be overloaded while others become idle, reducing the total traffic flow into the network.

In the artificial traffic load program, messages from each task are uniformly distributed among all other tasks. Thus, this program is a "perfect match" with the fully connected network with processor and communications integrated onto the same chip, since a direct link exists between each pair of communicating tasks. Because of the uniform traffic distribution, all links are equally utilized, and the amount of traffic generated by the processors is maximized. This rate determines the bandwidths shown in figure 3.8d.

Creating a new network by adding switching components increases the amount of traffic the network can carry, but the amount of traffic which can be generated is not increased. Thus, this additional network bandwidth cannot be utilized. In fact, performance will actually be degraded if the new network does not preserve the equal utilization of processor links described above. This phenomena explains the poor performance of some of the networks in figure 3.8d.

The behavior described above is atypical because the global/uniform traffic pattern of the artificial traffic load generator is not always appropriate. It will be seen that other topologies and different traffic patterns yield results favoring a small number of ports. Indeed, performance curves for the other application programs (figures 3.8a-b) indicate that networks using communication components with a small number of ports achieve better performance than networks with a large number of ports, even if the latter have the added advantage of including communication circuitry on the same chip as the processor. Networks with a small number of ports and processor and communications on the same chip will perform even better, widening this gap.

The curves for Barnwell's algorithm indicate that a significant performance improvement results from incorporating a multicast mechanism in the communication hardware. If no multicast mechanism is provided, the processor sending the message must send a separate copy to each destination. A queue appears instantly in the processor sending the message, leading to long delays and poor performance.

No multicast curve is shown when processor and communications are incorporated onto the same chip. This is because networks with and without a multicast mechanism behave identically under these circumstances. Since each processor has a direct link to every other processor, all "splitting apart" of the multicast packet is done at the source node. A network without a multicast mechanism behaves in exactly the same way for this topology.

### **3.6.2. Full-Ring Tree Networks**

The second topology is the full-ring binary tree [Desp78]. This topology is constructed from a binary tree by adding links between siblings and cousins, as shown in figure 3.9. The average hop count grows logarithmically with the number of nodes, while the number of branches per node remains fixed at 5.



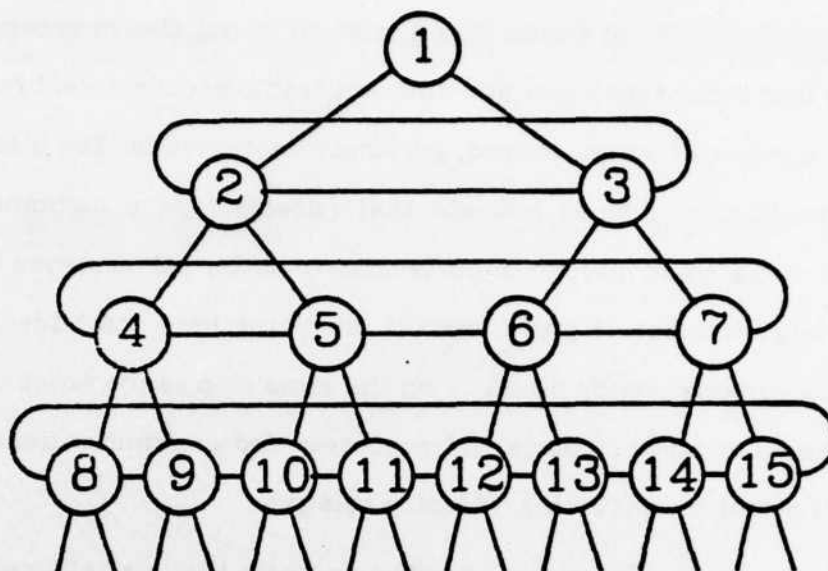
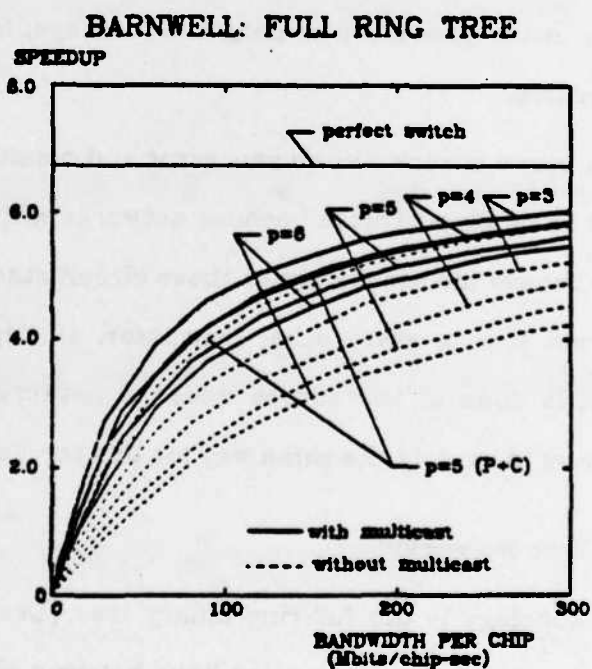


Figure 3.9. Full ring binary tree.



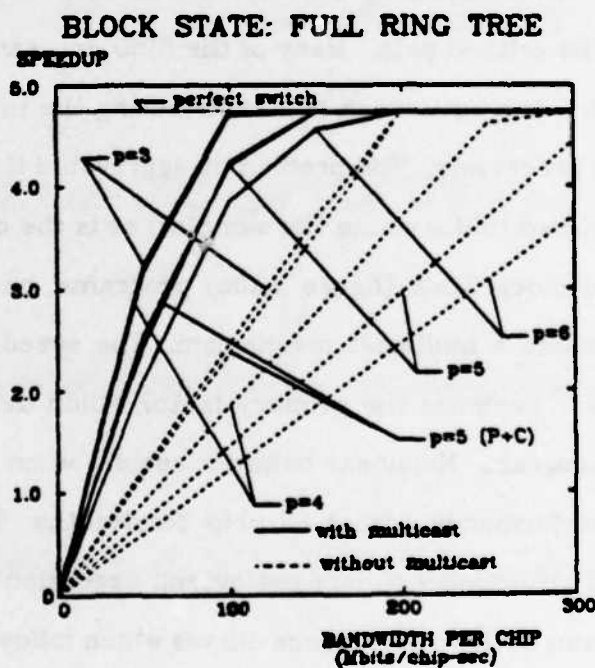
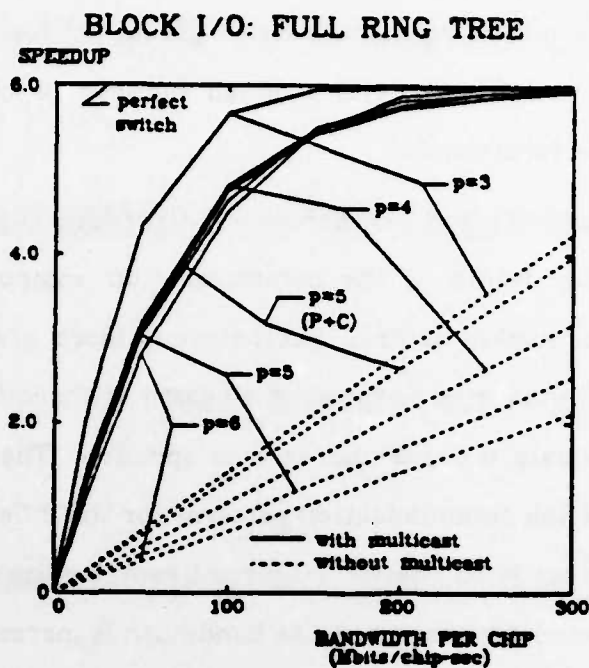
(a)

Figure 3.10. Full ring tree (a) Barnwell.

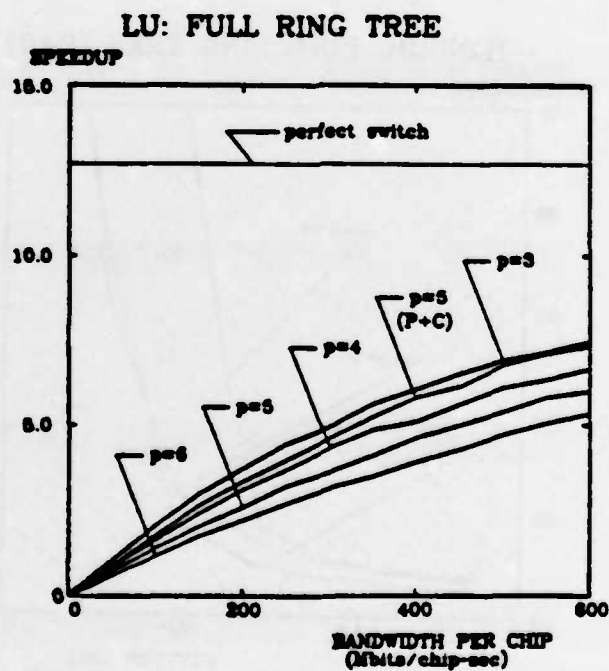
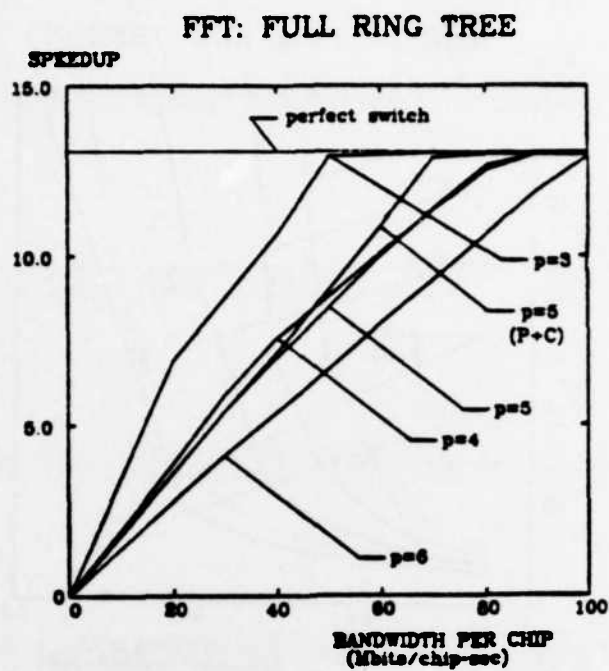
Performance curves for full-ring tree networks are shown in figures 3.10a-f. Qualitatively, these curves agree with those presented for the fully connected networks. Again, components with a small number of high-bandwidth links achieve the best performance.

The performance curves for Barnwell's algorithm (figure 3.10a) indicate that as the I/O bandwidth of the communication components is increased, speedup increases quickly at first, but becomes more gradual at higher link bandwidths. Other curves however, such as some of those for the FFT program (figure 3.10d), indicate a linear increase in speedup. These differences arise from the nature of the communication patterns for the different programs. The linear behavior arises when one virtual circuit remains the critical path for the program as chip bandwidth is varied. As bandwidth is increased, delay, and thus execution time, decrease in proportion. The pipelined programs often demonstrate this behavior, with the longest path from the first stage of the pipeline to the last forming the critical path. Many of the SISO programs also demonstrate this behavior. Here, the bottleneck is in distributing the initial data samples to the computations processors. The problem is aggravated if multiple-destination messages are required to distribute the samples, as is the case in the Block I/O (figure 3.10b) and Block State (figure 3.10c) programs, particularly if the network does not include a multicast mechanism. The speed of the links around the input processor becomes the primary factor which determines the execution time of the program. Nonlinear behavior results when no single virtual circuit dominates performance across all chip bandwidths. Instead, delays on a number of virtual circuits determine the overall execution time. Both types of behavior will be seen in the performance curves which follow.

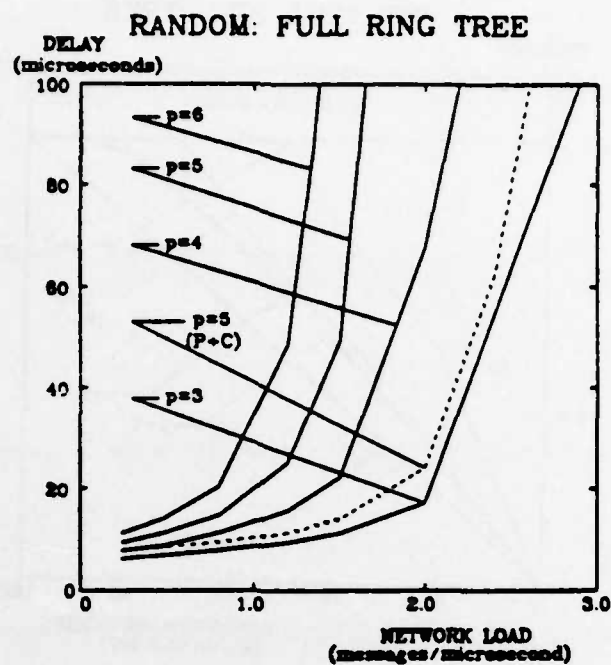
The curves for the artificial traffic load program (figure 3.10f) again indicate that delay and bandwidth are both improved as the number of ports is



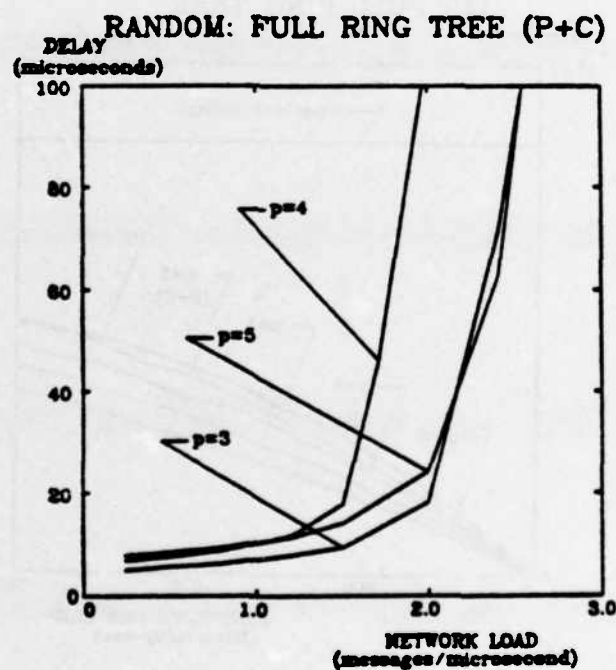
**Figure 3.10.** Full ring tree (b) Block I/O. (c) Block State.



**Figure 3.10.** Full ring tree (d) FFT. (e) LU.



(f)



(g)

Figure 3.10. Full ring tree (f) Random. (g) Random (P+C).

reduced. The curve with processor and communications integrated onto the same chip indicates that the link between the processor and communication component is still a bottleneck, since the bandwidth provided is significantly better than that provided by networks using components with 4, 5, or 6 ports per node. This bandwidth is still somewhat less than that of the Y-component network however, even though the latter is handicapped by this bottleneck link. This adds further support to components with a small number of ports.

### 3.6.3. Butterfly Networks

The 32 node butterfly network shown in figure 3.11 is the third topology studied. The butterfly is similar to the tree to the extent that the average hop count grows logarithmically with the number of nodes. Four branches are required for each node. This topology is more symmetric than the tree however, and thus is less susceptible to bottlenecks for applications exhibiting global traffic patterns. The butterfly is ideally suited for the FFT application program.

Performance curves for the butterfly network are shown in figures 3.12a-e. Due to the excessive amount of computing resources required, a curve for the Block I/O program could not be produced. Networks constructed with components using a small number of ports again achieve the best performance. The FFT program (figure 3.12c) performs unusually well at low chip bandwidths, demonstrating the reduction in bandwidth requirements when a good mapping is found between the application program and hardware. The curve with processor and communications on the same chip in the artificial traffic load program (figure 3.12e) indicates that the link between the processor and communication component is not a serious bottleneck. The bandwidth provided is equal to that of the network using communication components with the same number of ports.



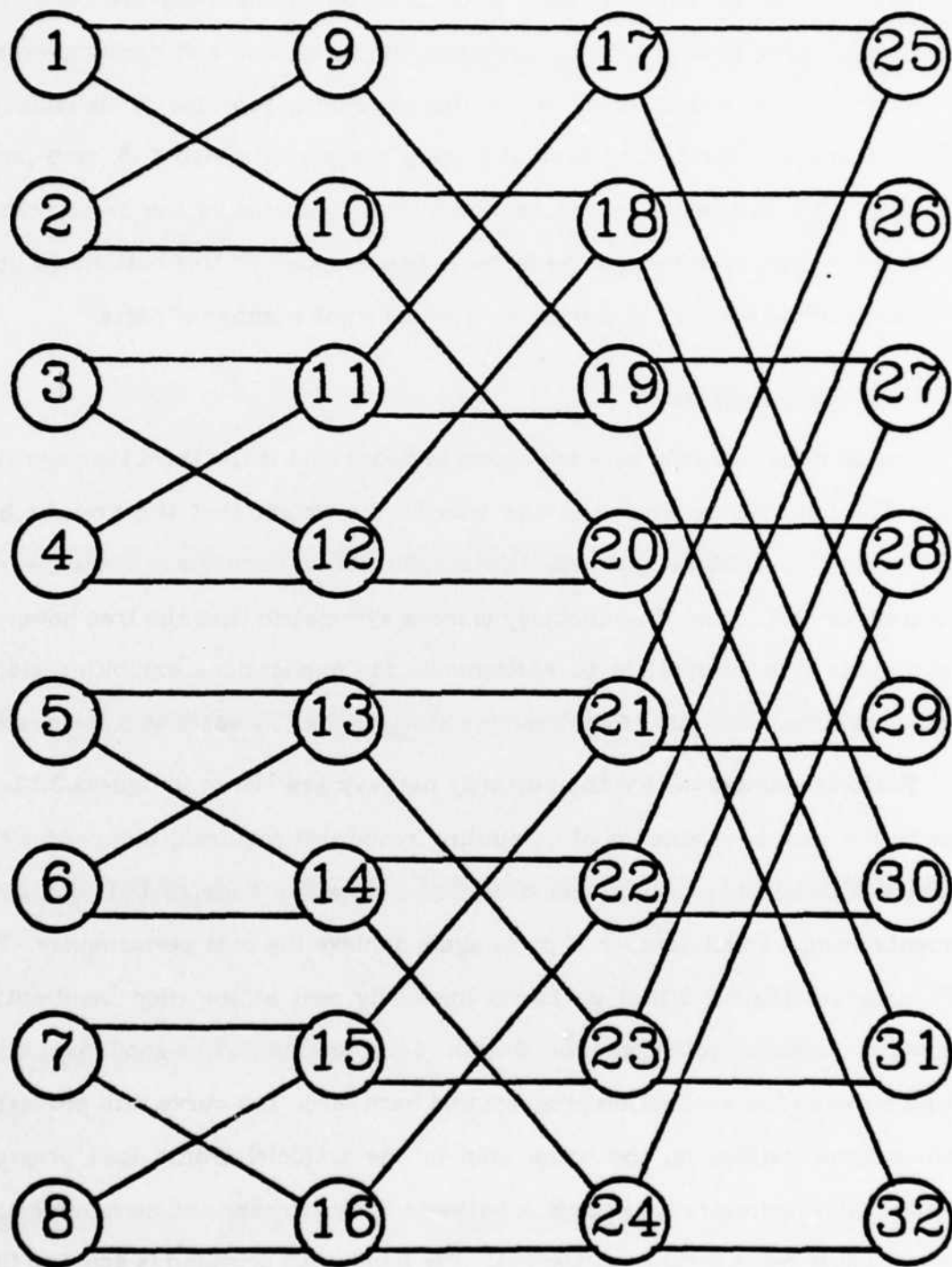


Figure 3.11. *Butterfly topology.*



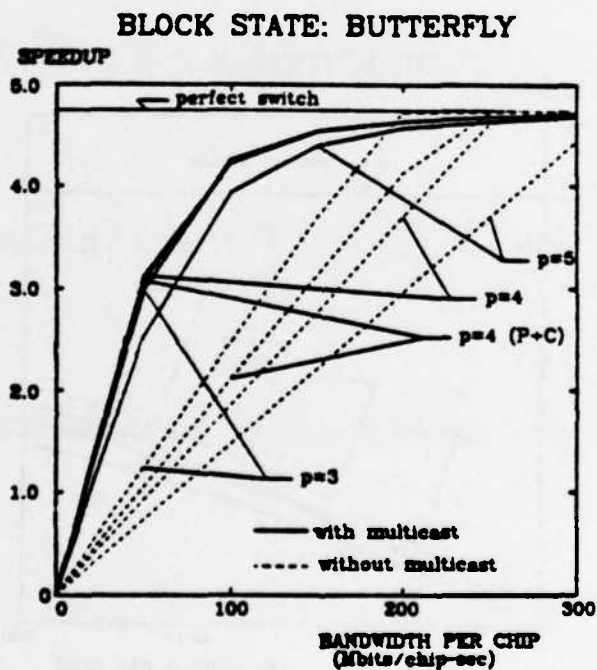
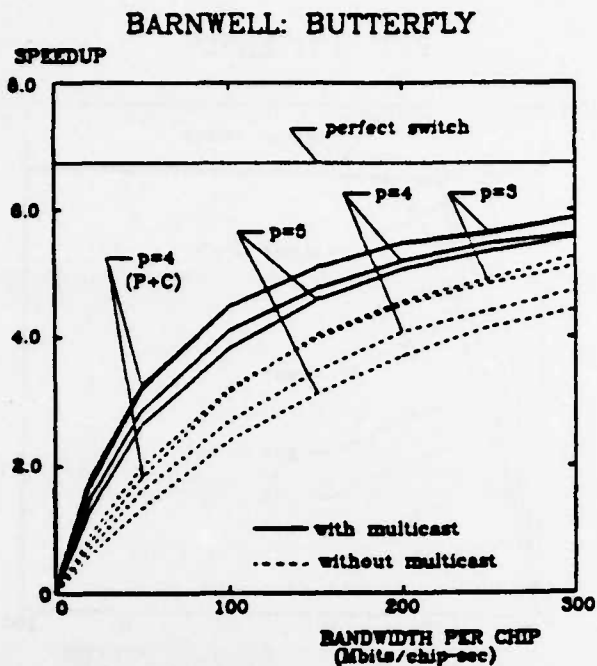


Figure 3.12. Butterfly (a) Barnwell. (b) Block State.

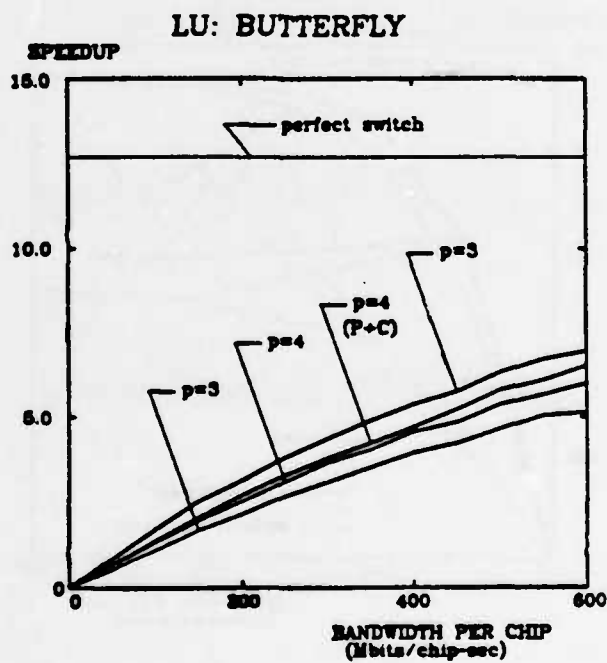
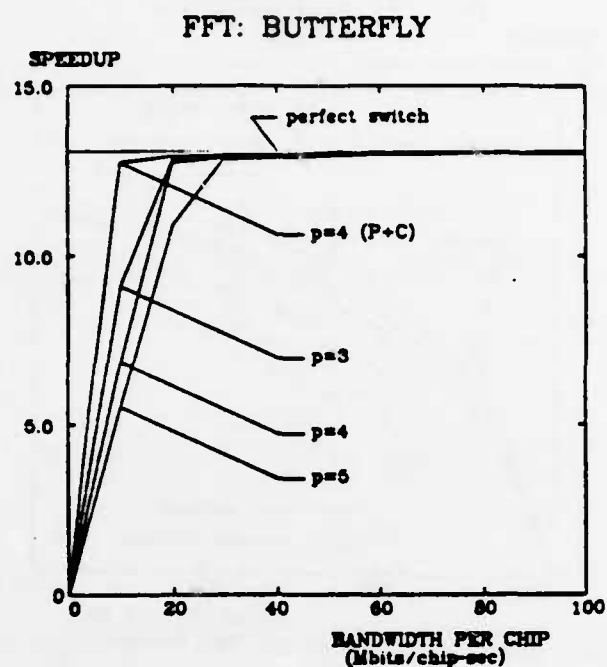
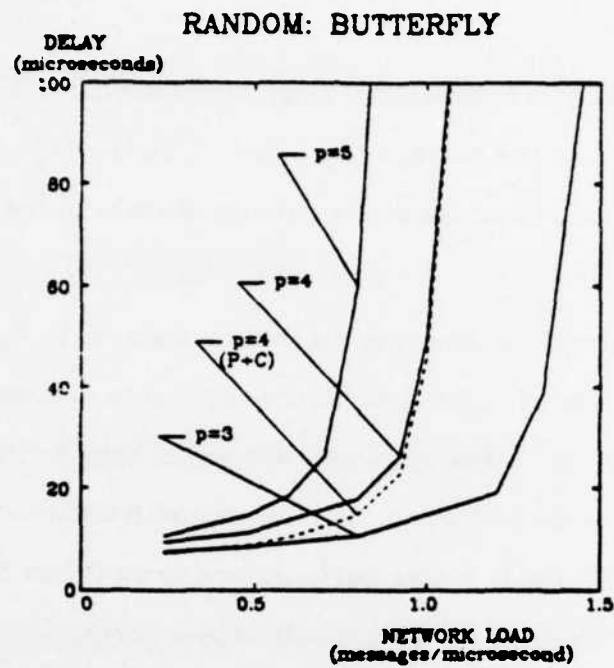
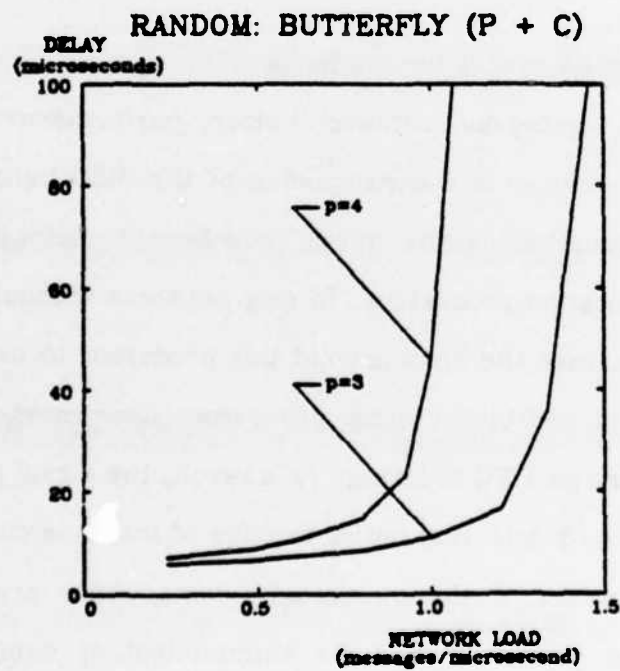


Figure 3.12. Butterfly (c) FFT. (d) LU.



(e)



(f)

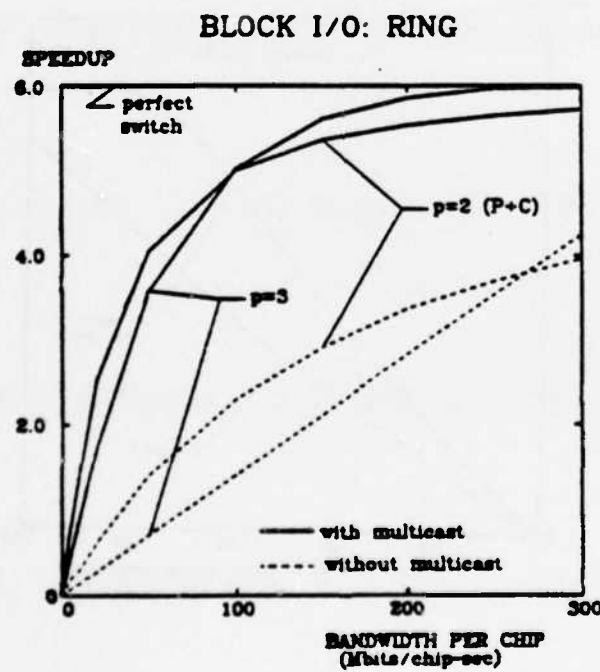
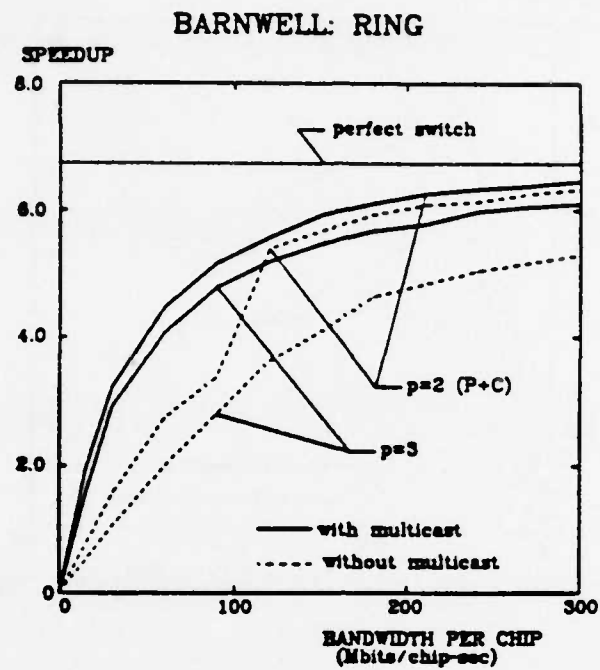
**Figure 3.12.** *Butterfly (e) Random. (f) Processor with Communications.*

#### 3.6.4. Ring Networks

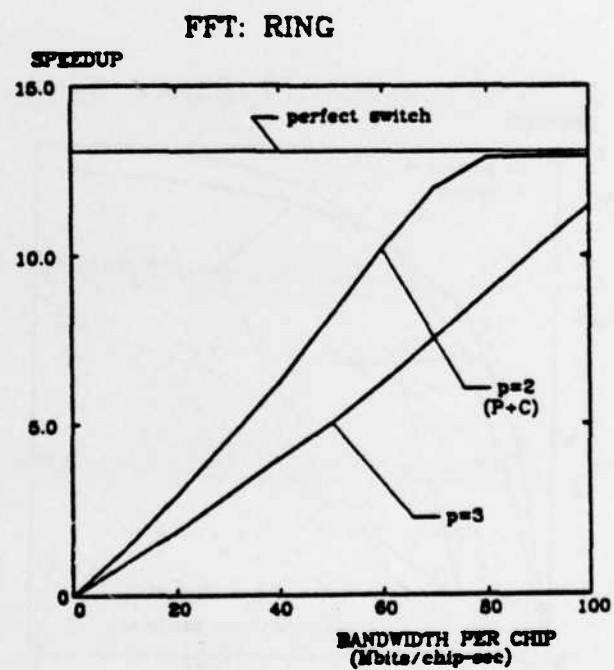
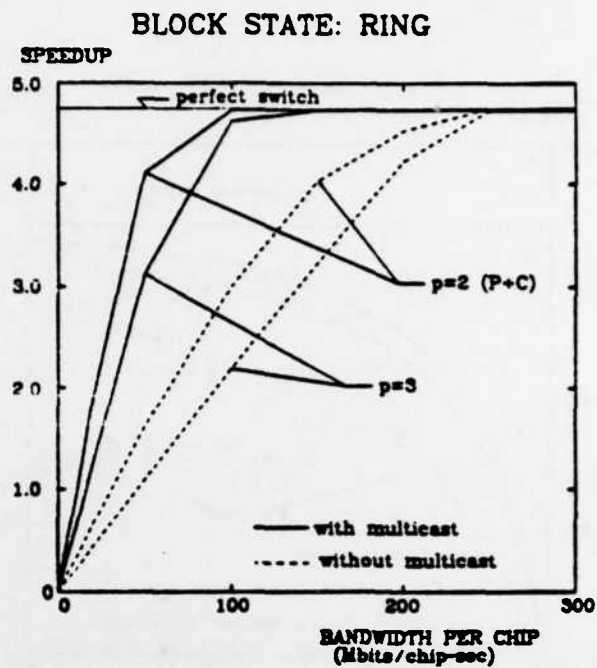
The fourth topology, a bidirectional ring, minimizes the number of branches per node, but maximizes the average hop count. Like the fully connected topology, the number of nodes is equal to the number of tasks in the application program.

Performance curves for the ring network are shown in figures 3.13a-f. In ring topologies, the use of communication components also implies the use of Y-components, since only three ports per chip are required. Thus, only two distinct networks need to be compared. The network with communication circuitry on the processor chip yields better performance since it has higher bandwidth links (only 2 ports are needed) and smaller hop counts. Thus, networks constructed with components with a small number of ports again achieve the best performance.

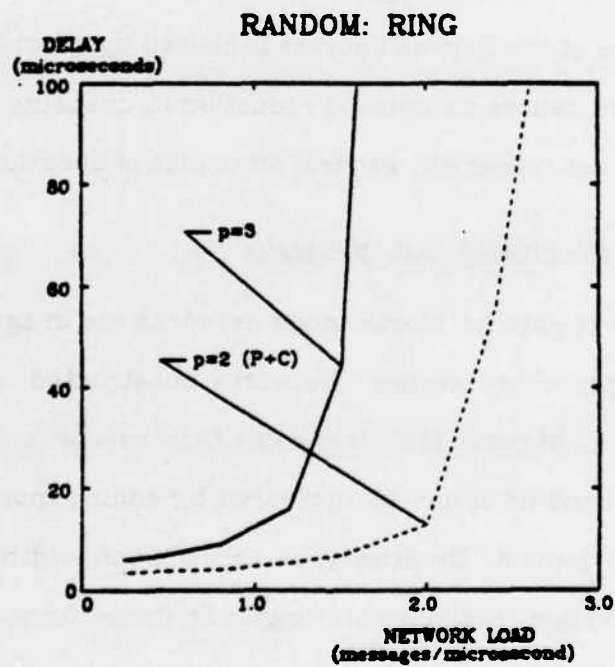
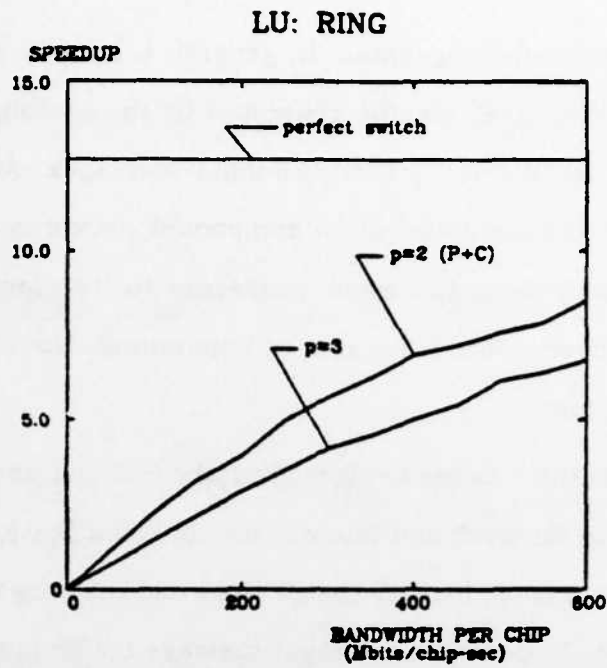
The only exception occurs for the Block I/O program. Here, the communication component networks achieve better performance at high chip bandwidths. This behavior is a consequence of the SISO behavior of the program. When execution begins, the "input" processor broadcasts data values to a number of computation processors. In ring networks without communication components, this causes the links around this processor to become saturated, blocking traffic produced by the other processors since messages are serviced at each node by a strict FIFO ordering. As a result, the signal processing calculations cannot proceed until this initial backlog of traffic is cleared up, slowing down the computation. If communication components are used, the link between the input processor and its communication component becomes saturated; however, this link does not block traffic among other processors. The arrival of the input data messages at the communication component is spread out over time. Thus, these messages do not completely block other traffic,



**Figure 3.13.** Ring (a) Barnwell. (b) Block i/o.



**Figure 3.13.** *Ring (c) Block State. (d) FFT.*



**Figure 3.13.** Ring (e) LU. (f) Random.



although they do increase congestion. In general, a priority mechanism could be used to avoid this anomaly: traffic generated by the computation processors can be assigned a higher priority than the input messages. At low bandwidths, the performance of the communication component networks is limited by the bandwidth of the link from the input processor to its communication component, allowing networks with processor and communications on the same chip to yield better speedup.

The "blocked traffic" behavior described above is not as prominent in the other SISO programs, Barnwell and Block State. In Block State, the traffic within the pipeline exhibits enough locality that it can avoid the congested area around the input processor. In Barnwell, the input message traffic is single destination, in contrast to Block State where the input traffic is multiple destination, and thus does not create as much congestion. The "jump" in performance around 110 Mbit/chip in one of the Barnwell curves is caused by a fortuitous shift in the traffic pattern which causes an unusual reduction in queueing delays along one of the links; it does not reflect any general principles of behavior.

### **3.6.5. Conclusions for Cluster Node Networks**

The simulation results for cluster node networks are in agreement with the analytical results presented earlier. Networks constructed from components using a small number of ports yield less delay than networks using components with many ports. Bandwidth can be increased by adding more components to the communication domain. Eventually, as network bandwidth is increased, the rate at which processors can generate traffic limits performance, rather than the bandwidth of the network. Also, the programs using multiple-destination communications show a significant performance improvement if the communication circuitry includes a multicast mechanism.

### 3.7. Simulation Results on Networks with a Fixed Number of Components

Cluster nodes constructed from components with a large number of ports require fewer components than those constructed with a small number of ports. Thus, the studies presented above do not consider chip count. In this section, networks using the same number of components are considered. The application programs are executed on lattice and tree topology networks like those analyzed earlier. It will be seen that bottlenecks form around the root of the tree networks, biasing the results to favor components with a small number of high bandwidth links. De Bruijn networks are examined as an example of a class of network topologies with logarithmic average hop count, but without this inherent bottleneck.

The analytical results indicated that networks constructed from components with a small number of ports yielded lower delay, but less bandwidth than networks using components with a large number of ports. Based on these results, one would expect networks using a large number of ports to yield better performance when the network is bandwidth limited. Intuitively, as we move toward networks with a larger number of (slower) links, the average hop count is reduced, and additional paths are created in the network. These trends combine to reduce traffic on congested links. If the reduction in congestion is significant, it will more than offset the disadvantage of using slower links, and overall performance improves. Of course, if the network provides adequate bandwidth for the traffic load presented to it, then the queueing delays will be small, and networks using a larger number of ports can only achieve poorer performance since link speed is reduced. Thus, networks with a large number of ports can be expected to provide better performance when the traffic load is heavy relative to total network bandwidth, but networks with a small number of ports can be expected to perform better otherwise.

### 3.7.1. Lattice Topologies

The application programs were run with switch models for the lattice topologies shown in figure 3.14a-c. Performance curves are shown in figures 3.15a-f. The FFT program exhibits better performance with a large number of ports, as would be expected in bandwidth-limited networks. The remaining programs however, indicate little performance variation as the number of ports is varied, or better performance with a small number of ports. One reason for this is that most of the programs encounter bottlenecks which are not alleviated when the number of ports, and thus the number of paths through the network, is increased. In the SISO programs for example, the bottleneck is around the input processor, and performance is determined to a large extent by the speed of the communication links around this congested area. Since components with a small number of ports use faster links, they achieve better performance.

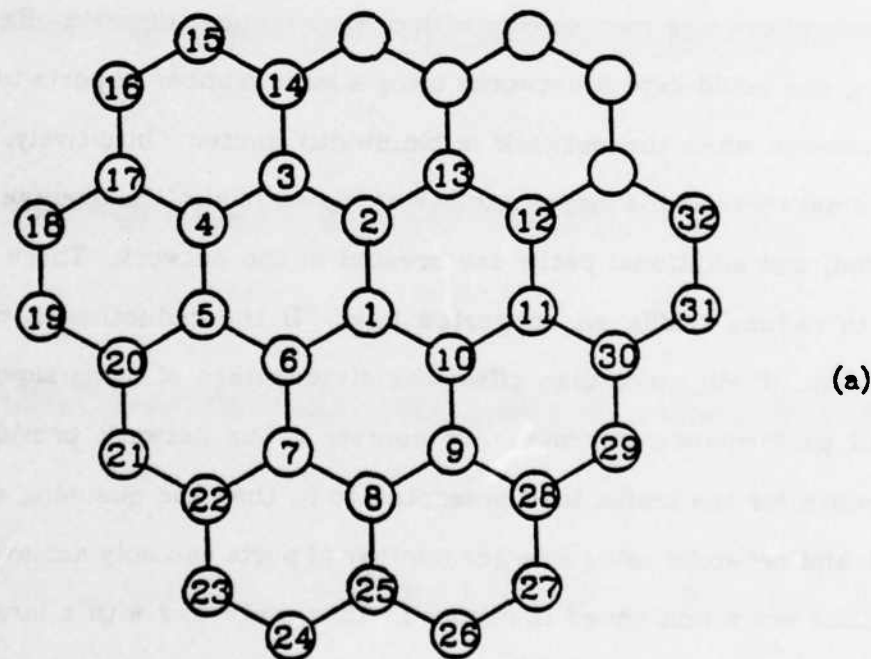
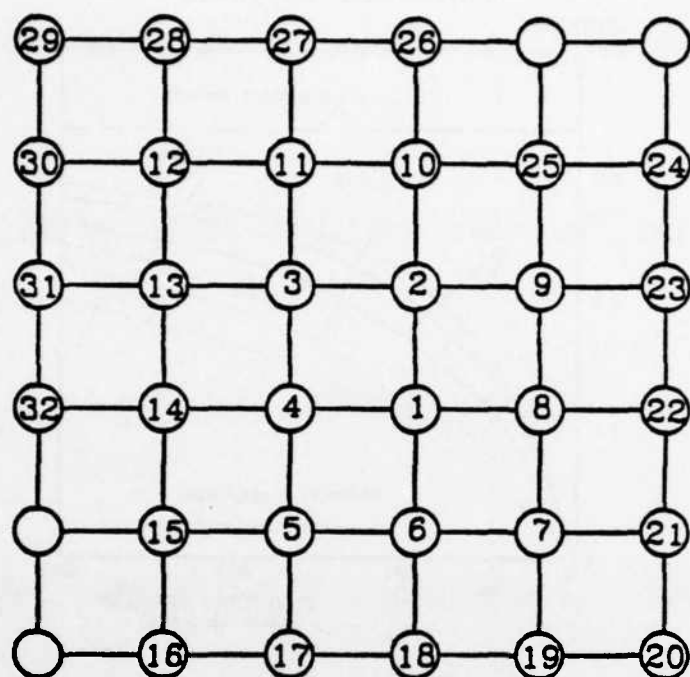
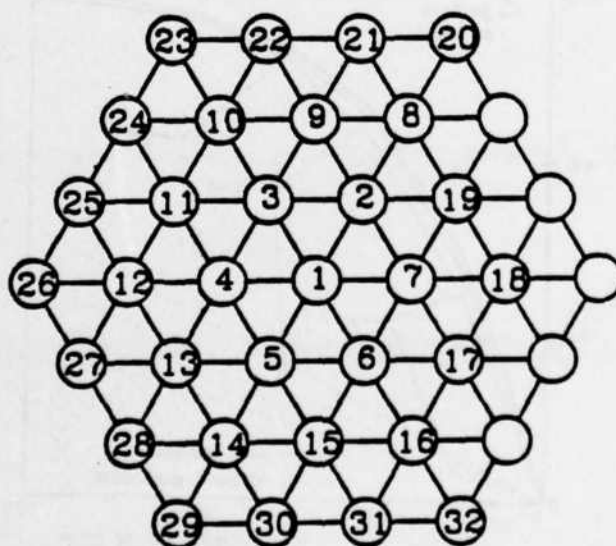


Figure 3.14. Lattices (a) 3 ports.



(b)



(c)

**Figure 3.14.** Lattices (b) 4 ports. (c) 6 ports.

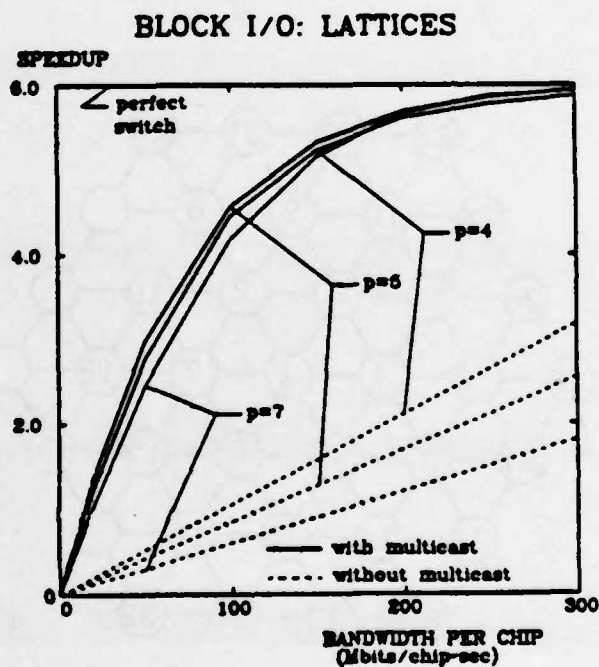
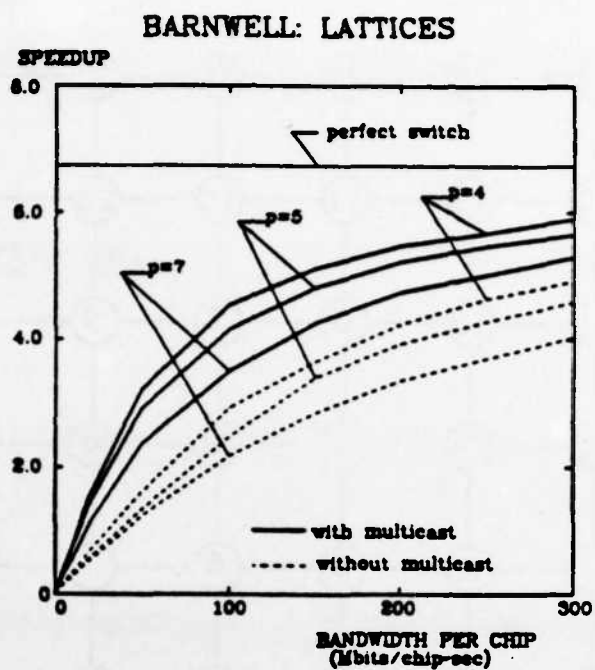


Figure 3.15. Lattices (a) Barnwell. (b) Block I/O.

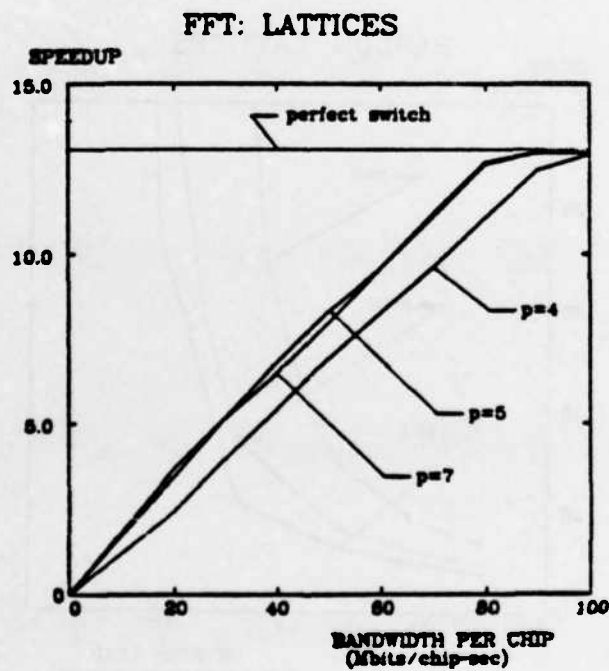
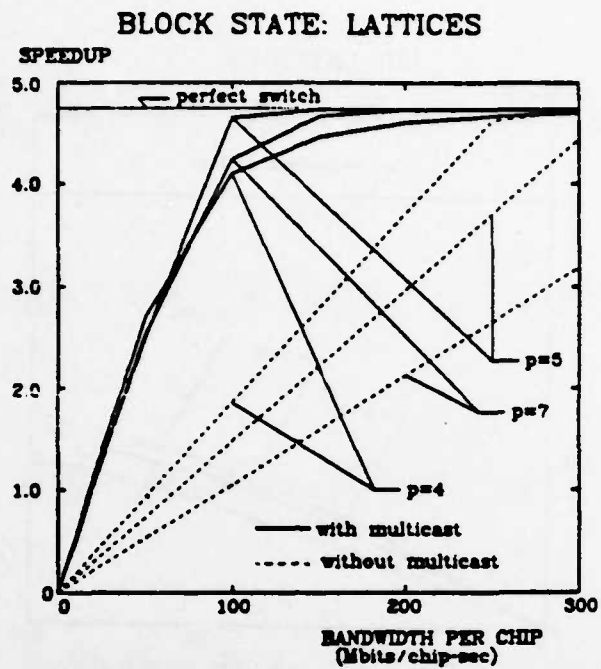


Figure 3.15. Lattices (c) Block State. (d) FFT.



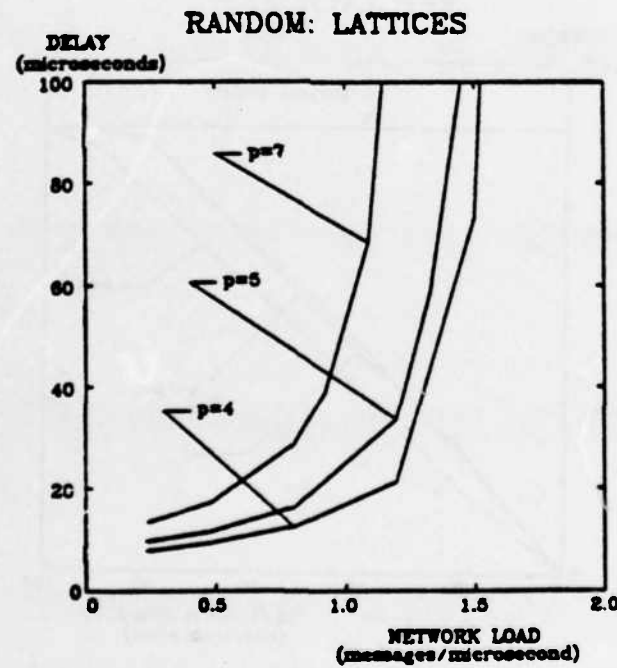
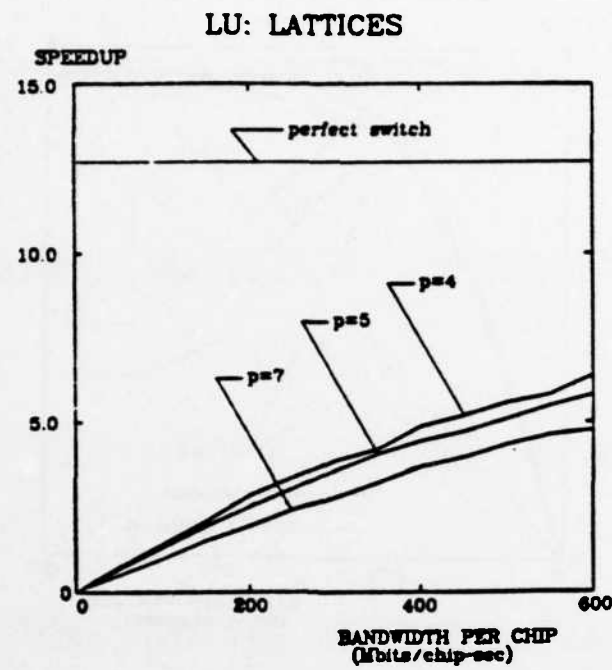


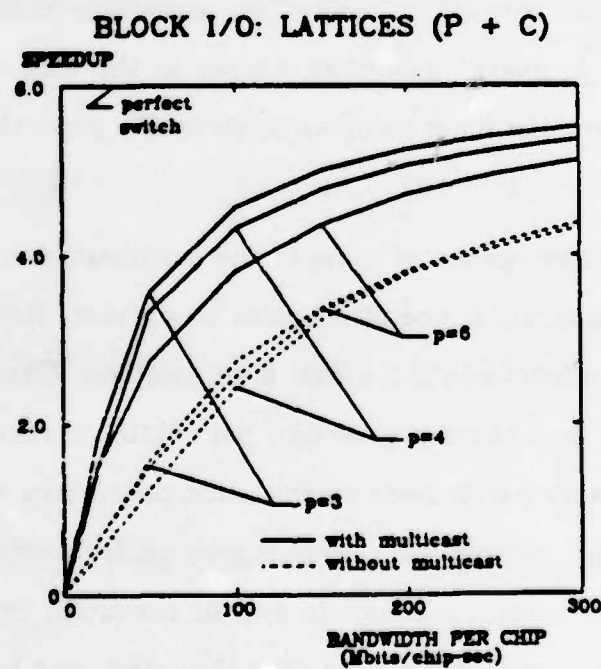
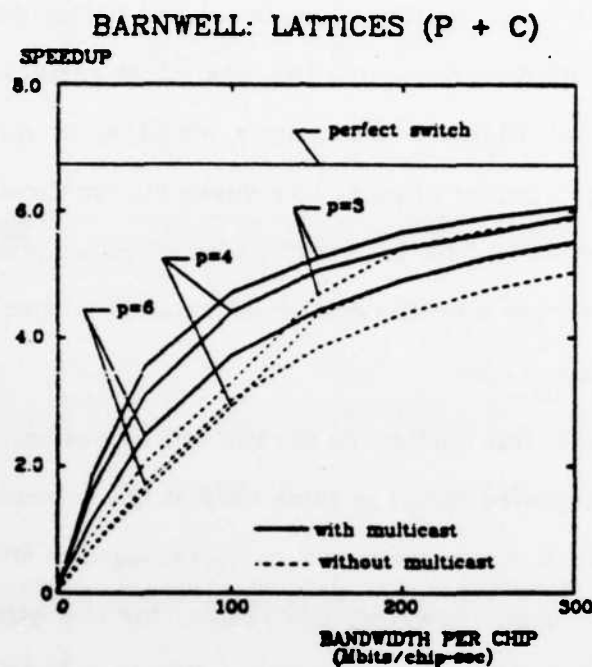
Figure 3.15. Lattices (e) LU. (f) Random.



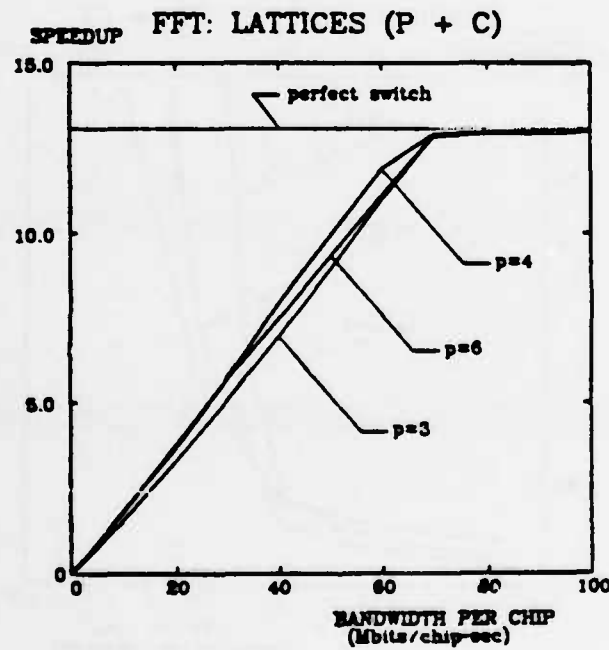
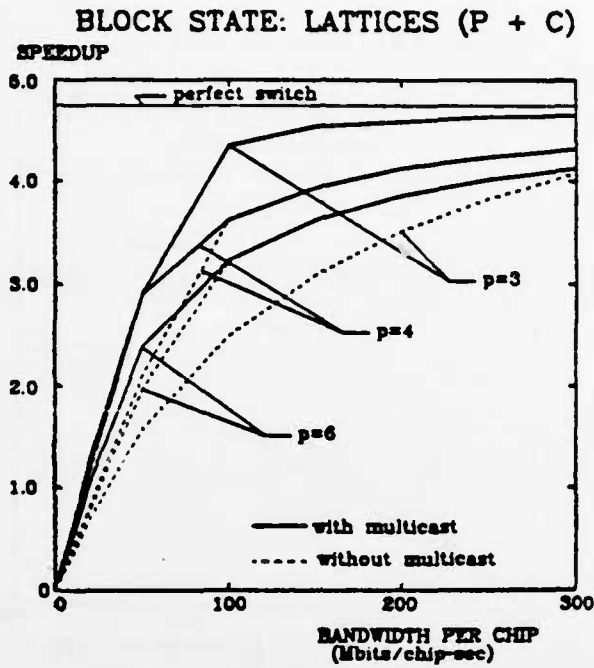
The delay/bandwidth curves for the artificial traffic load program indicate that networks with a small number of ports achieve better delay and bandwidth. The bandwidth result disagrees with the analytical results presented earlier, which suggested that reduced hop counts would allow networks using components with a large number of ports to achieve higher throughput. The reason for the disagreement is that for high traffic loads, the processor/communication component link becomes a bottleneck. Performance is thus determined by the speed of this bottleneck link.

In figures 3.16a-f, this bottleneck is removed by assuming that communication circuitry is integrated onto the same chip as the processor. The curves for the artificial traffic load program are in closer agreement with the analytic results presented earlier, however, the results for the other application programs are qualitatively the same. It is interesting to note that some of the SISO programs, Block State and Block I/O in particular, experience lower performance when this latter model is used. This anomalous effect can be attributed to the "blocking problem" described earlier in the ring topology discussion. Messages that carry the input samples block traffic generated by the computation processors.

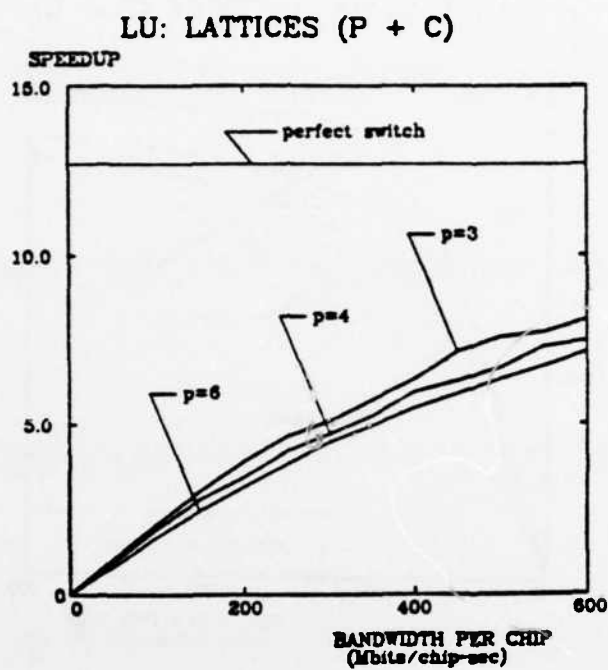
Finally, the convergence of some of the multicast/non-multicast curves in the Block State program is one other point of interest. Recall that Block State uses multicast to distribute the initial data samples. The convergence of the curves indicates that beyond a certain bandwidth, here approximately 100M bit/chip, the network can provide computation processors with data samples as quickly as they can be processed, so improved performance along these virtual circuits results in no improvement in overall execution time. In addition, the network provides enough bandwidth that the additional traffic caused by the absence of a multicast mechanism does not degrade performance.



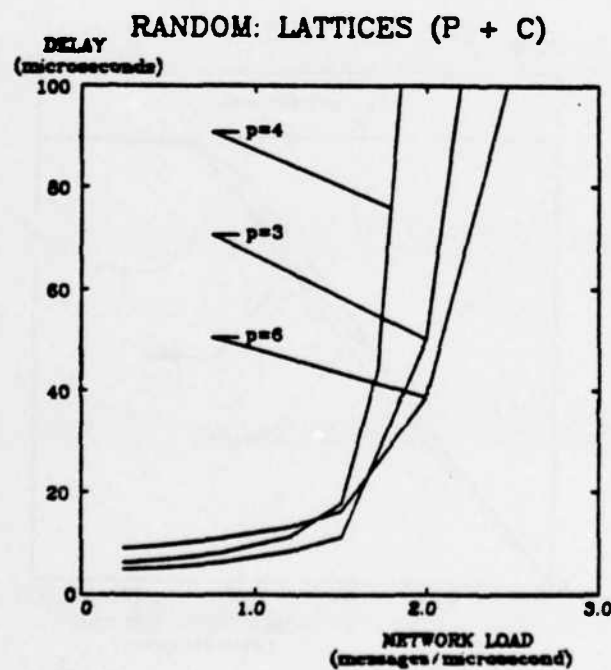
**Figure 3.16.** Lattices (P+C) (a) Barnwell. (b) Block I/O.



**Figure 3.18.** *Lattices (P+C) (c) Block State. (d) FFT.*



(e)



(f)

Figure 3.16. Lattices (P+C) (e) LU. (f) Random.

### 3.7.2. Tree Topologies

Performance curves for tree networks (figure 3.17 shows one such network) are shown in figures 3.18a-g. For all application programs, it is seen that networks built from components with a small number of ports yield better performance than those using a larger number of ports, even when processor and communications are incorporated onto the same chip (see figure 3.18g). These results however, are a consequence of congestion around the root node rather than from the hop count/link bandwidth tradeoffs discussed earlier. In trees, a disproportionate amount of traffic must flow through the root, leading to congestion in this portion of the network. Increasing the number of links does not improve the amount of bandwidth allocated to this congested area. As a result, performance is determined to a large extent by the speed of communication links near the root. Since components with a small number of ports have faster links, they yield higher performance.

### 3.7.3. De Bruijn Networks

The results for tree topologies were biased because of the inherent bottleneck around the root. To provide a true test of the analytical results, a class of topologies is required which does not have this inherent bottleneck, but which also has an average hop count which grows logarithmically with the number of nodes. The class of topologies must be general to the extent that networks with approximately the same number of nodes can be constructed as the number of ports is increased.

One class of topologies which satisfy these requirements are De Bruijn networks [Brui46]. De Bruijn networks, which are only defined for even degree (i.e. an even number of links per node), are the densest known infinite family of undirected graphs of even degree greater than 4. A dense graph of degree  $p$  is one with a small diameter. Diameter, which is specified as a function of  $p$  and

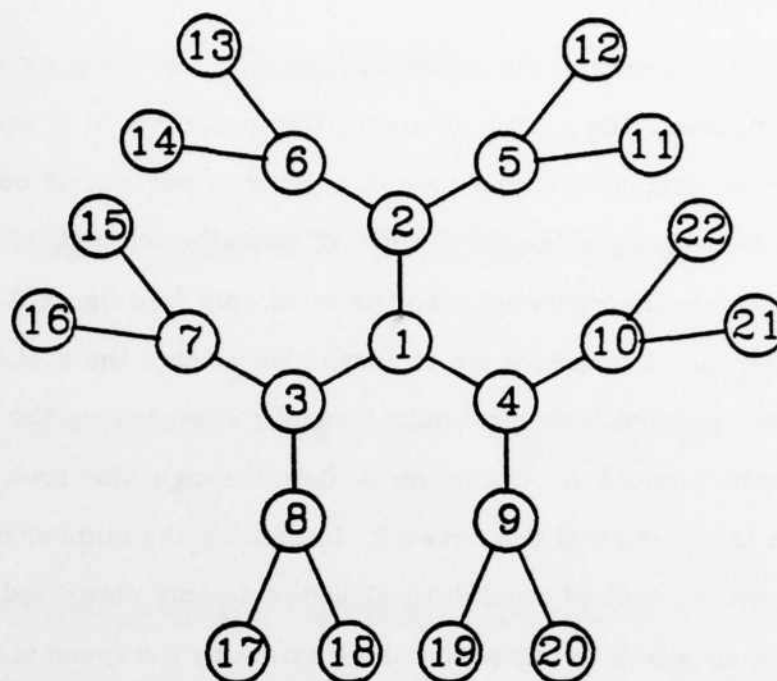


Figure 3.17. Tree topology.

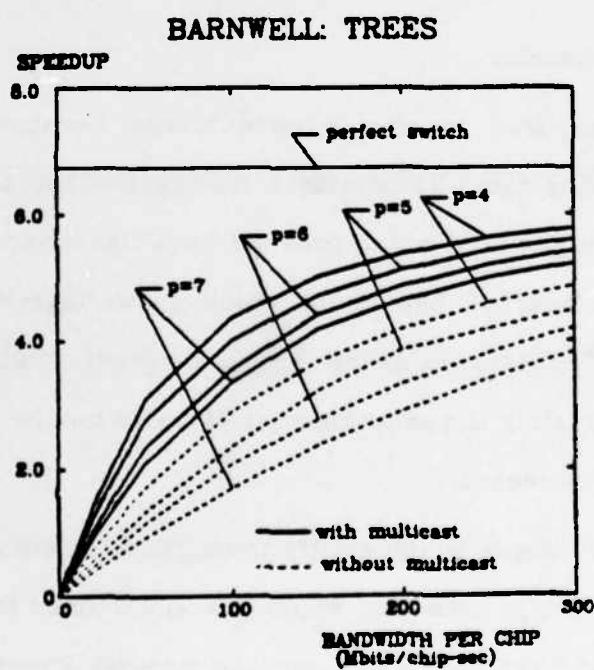
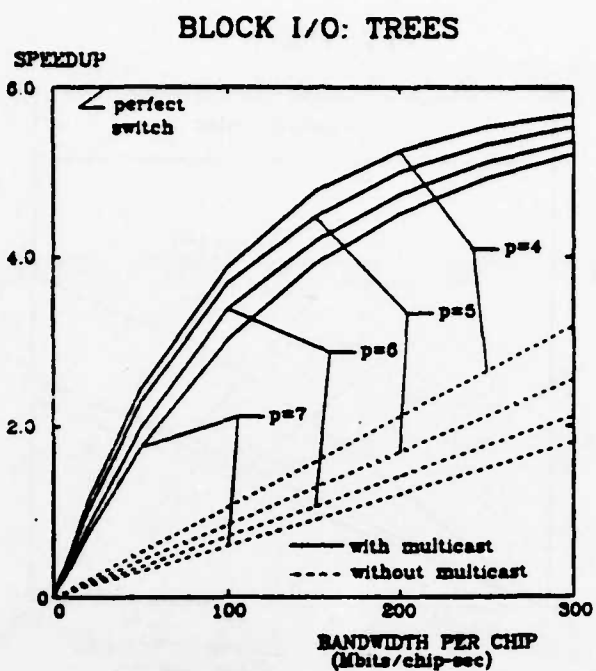
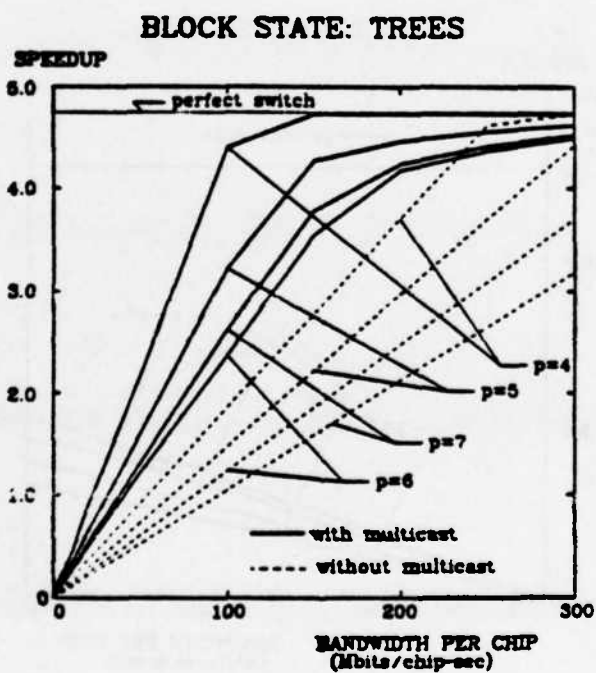


Figure 3.18. Trees (a) Barnwell.



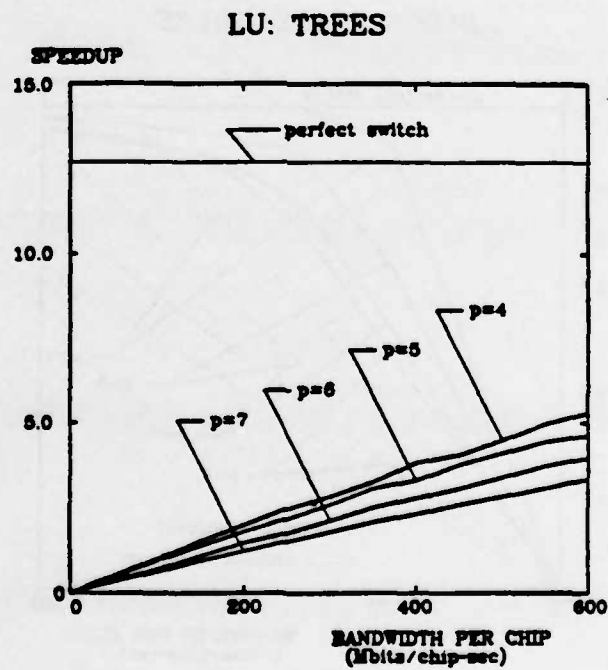
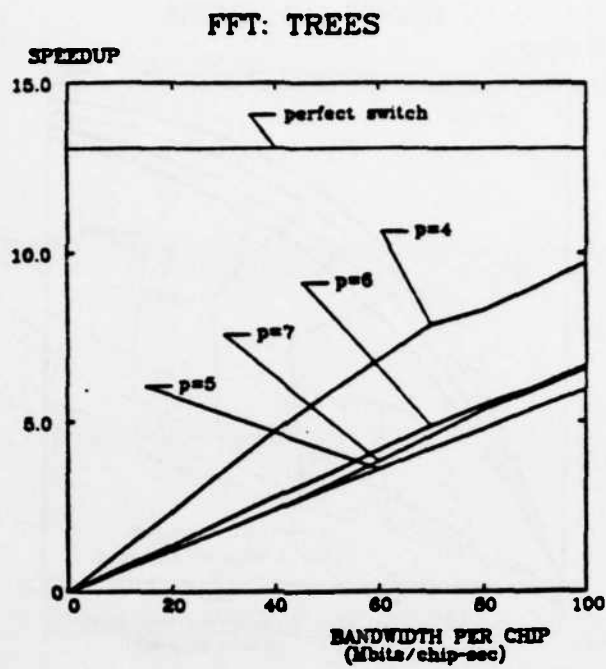
(b)



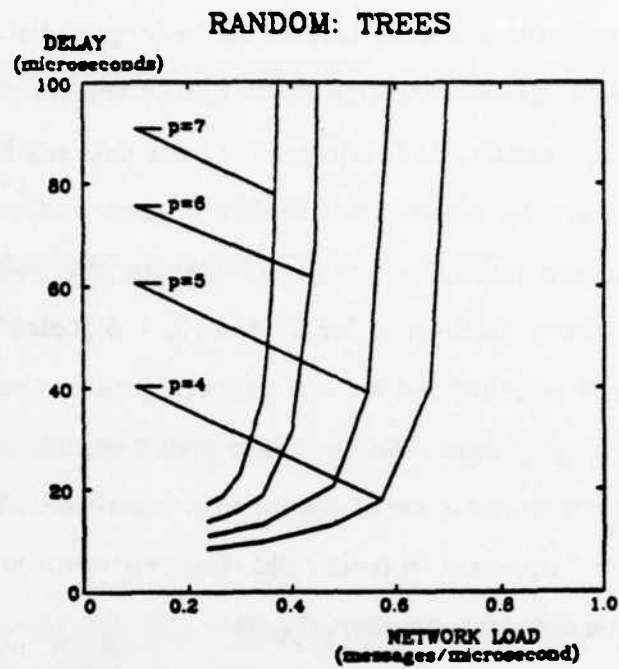
(c)

Figure 3.18. Trees (b) Block I/O. (c) Block State.

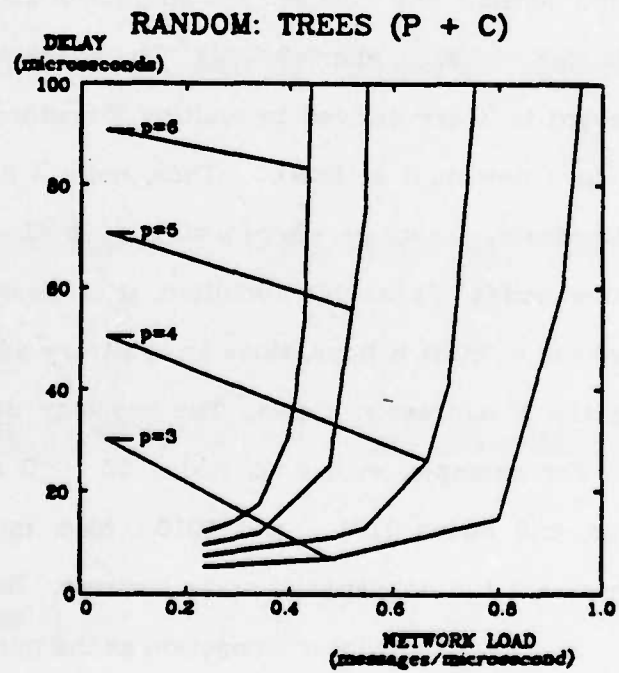




**Figure 3.18. Trees (d) FFT. (e) LU.**



(f)

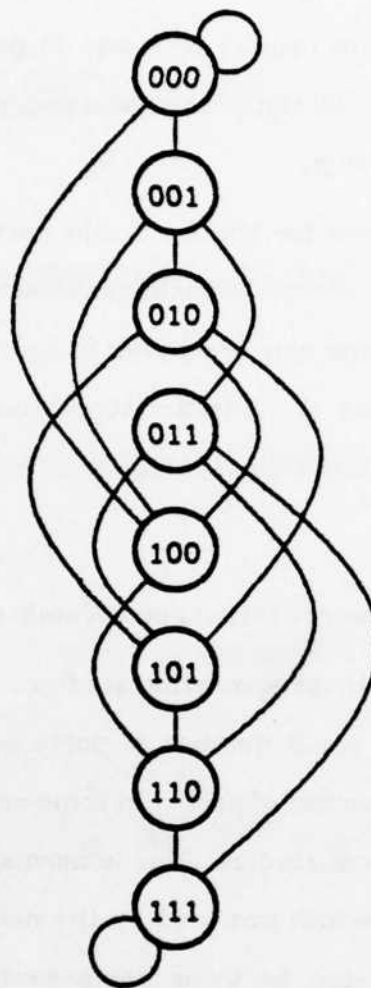


(g)

Figure 3.18. Trees (f) Random. (g) Processor with communications.

the number of nodes in the graph, is defined as the largest distance between any pair of nodes, where distance refers to the length of the shortest path between the two nodes. Until recently, De Bruijn graphs were not only the densest family of graphs of degree greater than 3, but De Bruijn graphs of degree  $p$  were also denser than any other family of graphs of degree  $p+1$ . Recently however, denser graphs have been discovered for degrees 3, 4, 5 [Lela82a, Lela82b]. Also,  $C_5'$  graphs, which are only defined for odd degrees greater than 3, yield smaller diameter than De Bruijn graphs with one fewer port per node [Farh81]. Still, the De Bruijn networks represent a set of graphs with logarithmic hop count without the "root bottleneck" inherent in trees, and thus represent an attractive topology for analyzing the optimum number of ports.

A De Bruijn graph is characterized by two parameters, a base  $b$  and an integer  $n$ . The graph consists of  $b^n$  nodes. The address of each node is defined by a string of digits,  $x_0x_1 \cdots x_{n-1}$ , where  $0 \leq x_i < b$ . The addresses of nodes which are directly connected to  $X$  are derived by shifting  $X$ 's address left or right 1 digit, and shifting in a new digit  $k$ ,  $0 \leq k < b$ . Thus, node  $X$  has links to nodes  $yx_0x_1 \cdots x_{n-2}$  and nodes  $x_1 \cdots x_{n-1}y$ , where  $y=0, 1, \cdots, b-1$ . Each node has up to  $2 \times b$  links to other nodes. From this definition, it is clear that node  $X$  can reach any other node in at most  $n$  hops, since an arbitrary address can be generated by shifting the  $X$  address  $n$  times. The topology does contain some degenerate cases. For example, with  $b=2$ , nodes  $00 \cdots 0$  and  $11 \cdots 1$  have links to themselves, and nodes  $0101 \dots$ , and  $1010 \dots$  have more than one link between them. These are the only special cases however. The edges of the De Bruijn graph yield exactly the same interconnection as the permutation network sometimes called the single-stage shuffle-exchange [Ston71, Ston72]. A base 2, 8 node network is shown in figure 3.19.



**Figure 3.19.** *Base 2 De Bruijn network.*

For this study, three De Bruijn graphs were examined:

- (1)  $b=2, n=5$  (32 nodes)
- (2)  $b=3, n=3$  (27 nodes)
- (3)  $b=5, n=2$  (25 nodes).

These graphs were selected since they have roughly the same number of nodes, and also provide enough processors to execute most of the application programs (the FFT is the only one requiring more than 25 processors). Communication

components for these graphs require 5, 7, and 11 ports for each node, respectively, including one port to attach to the node's computation processor, providing a wide range in values for  $p$ .

The performance curves for the De Bruijn networks described above are shown in figures 3.20a-e. Performance with processor and communication circuitry integrated onto the same chip are shown in figures 3.21a-e. The results are qualitatively similar to those of the lattice topologies. The curves indicate that better performance is achieved when components with a small number of ports are used.

#### **3.7.4. Conclusions for Networks with a Fixed Number of Components**

The primary result of these simulation studies is that networks constructed with components using a small number of ports achieve better performance than those using a large number of ports. In some cases, this is in disagreement with the results of analytical studies. This is normally due to bottlenecks that prevent much of the bandwidth provided by the network to be utilized. These bottlenecks can be alleviated by using components with a small number of ports, since this provides maximum bandwidth for the concerned links. The bottlenecks may arise from the application program (e.g. the SISO programs here), or from the network topology (e.g. trees). The limited I/O bandwidth of the processors generating messages may be the source of another bottleneck. Finally, the simulations also demonstrate that significant performance improvements can be achieved if mechanisms are included for efficient handling of multiple-destination messages.

#### **3.8. Influence of the Mapping of Tasks to Processors**

The results described above assumed a specific algorithm, to be discussed below, for mapping application programs onto the network topologies. Care

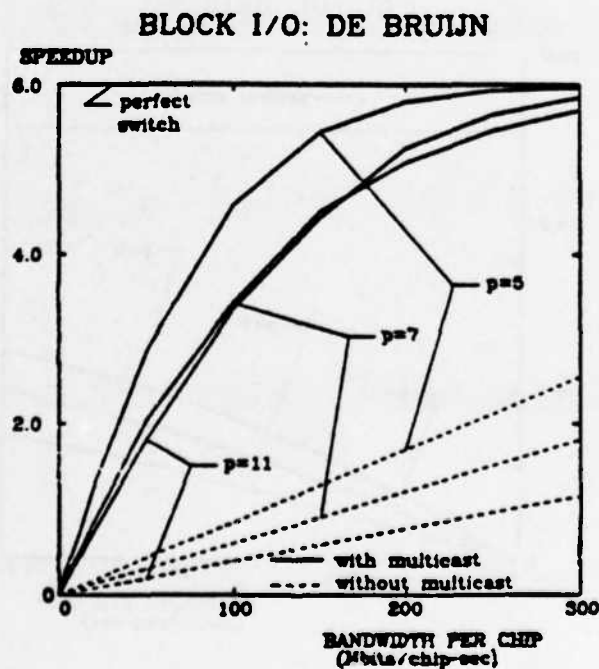
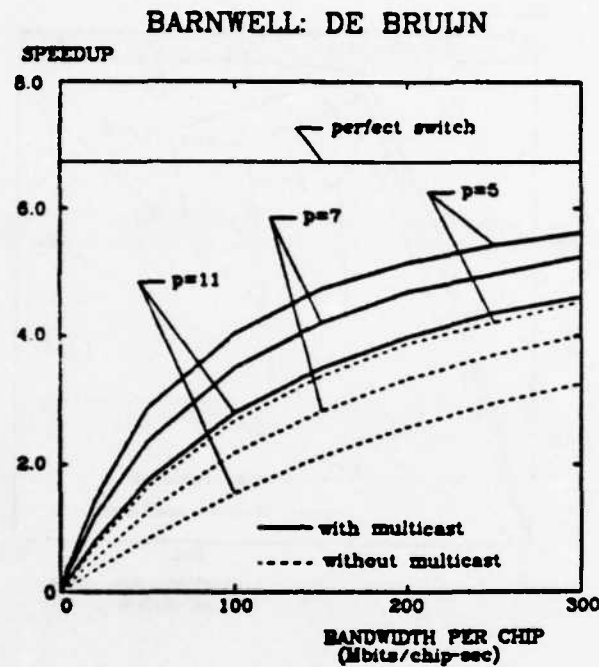


Figure 3.20. De Bruijn network (a) Barnwell. (b) Block I/O.

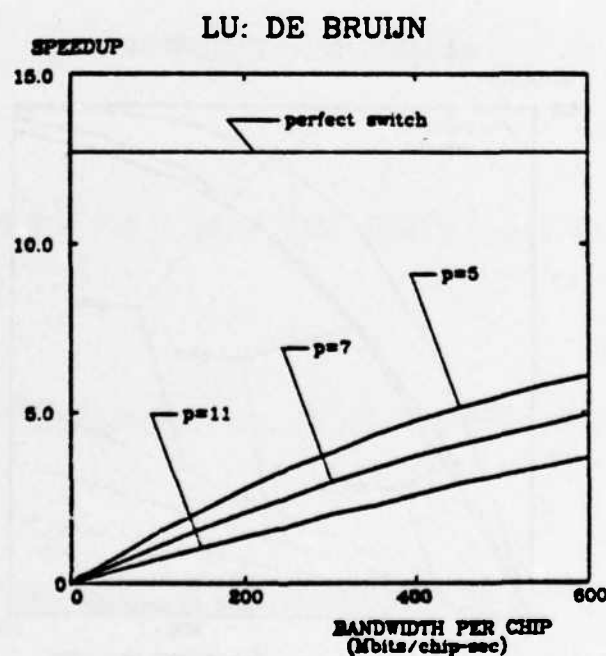
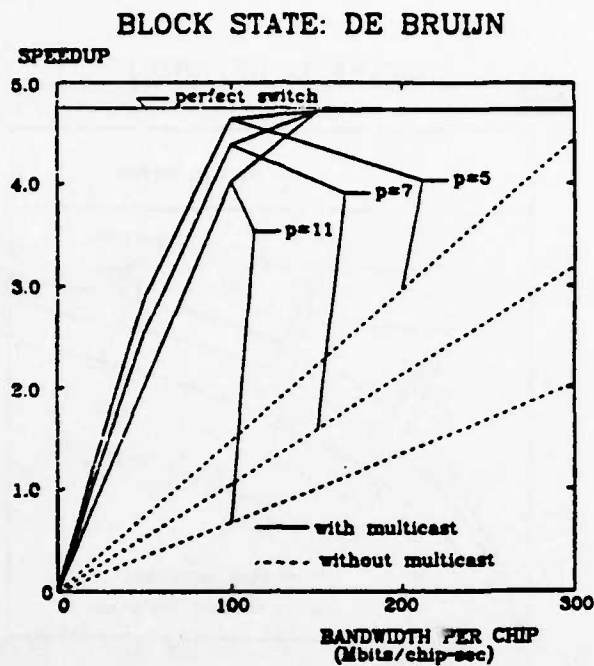
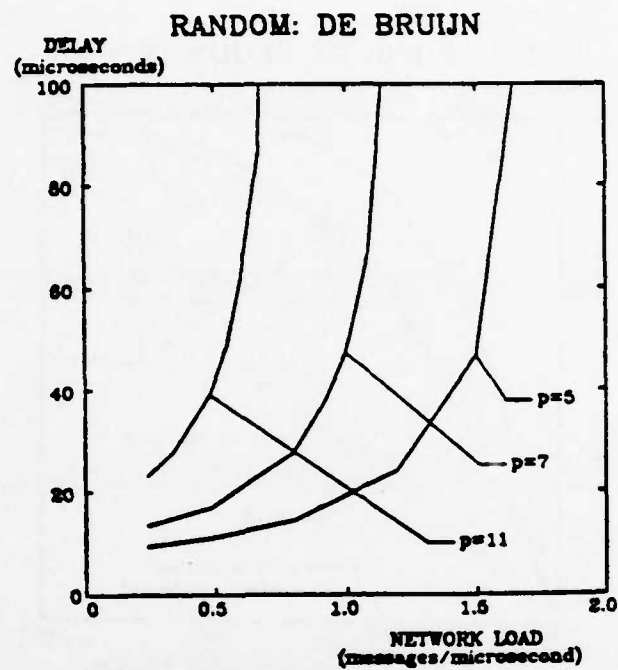


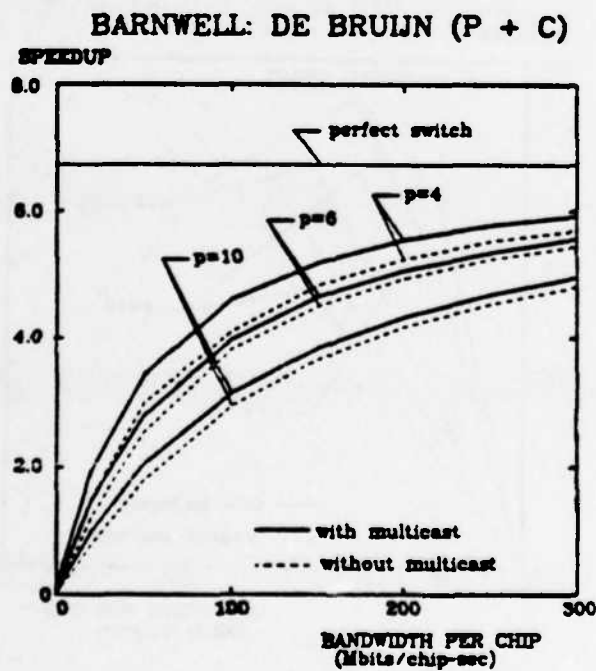
Figure 3.20. De Bruijn network (c) Block State. (d) LU.





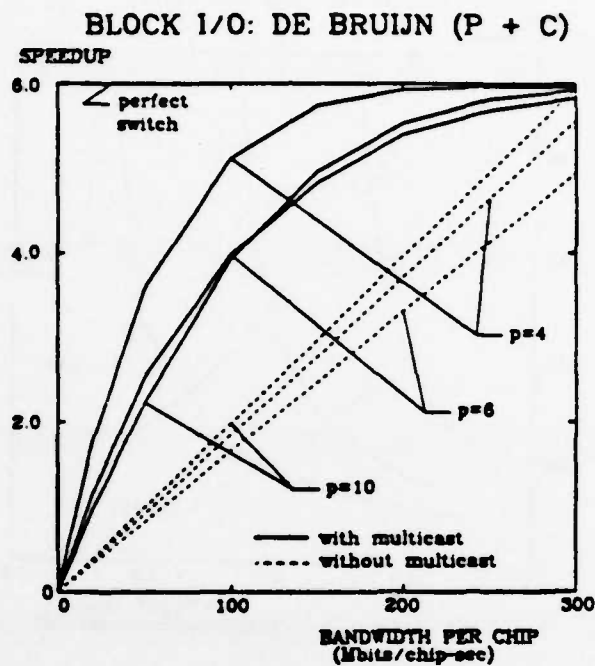
(e)

Figure 3.20. De Bruijn network (e) Random.

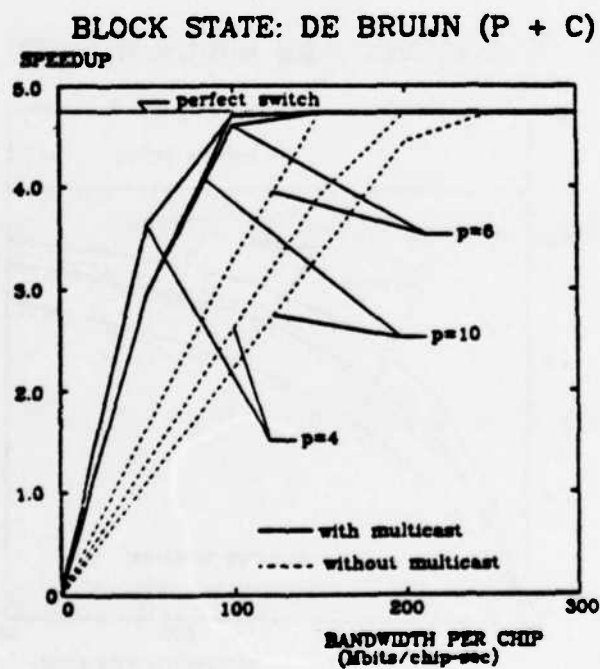


(a)

Figure 3.21. De Bruijn network (P+C) (a) Barnwell.



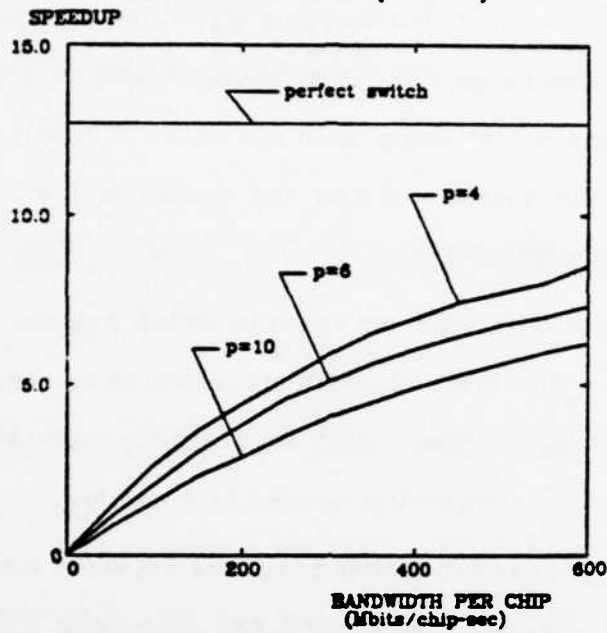
(b)



(c)

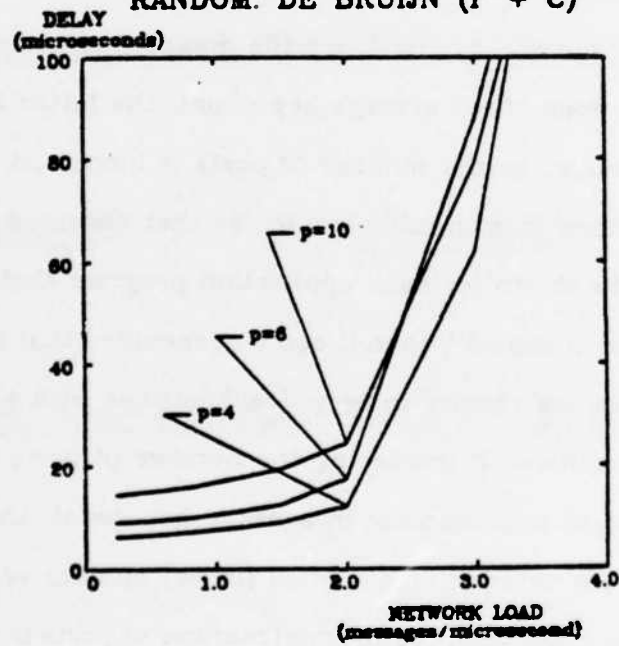
**Figure 3.21.** *De Bruijn network (P+C) (b) Block I/O. (c) Block State.*

## LU: DE BRUIJN (P + C)



(d)

## RANDOM: DE BRUIJN (P + C)



(e)

Figure 3.21. De Bruijn network (P+C) (d) LU. (e) Random.

must be taken to ensure that this mapping algorithm is "equally good" for the networks being compared, or else the differences between the curves may just be a result of using a better mapping in one network relative to another. This section addresses the question of how the quality of the mapping algorithm affects the results presented above.

The simulation results used two types of switch models. The first is based on the cluster node. However, cluster node networks are only an implementation of a given topology. Thus, within each topology, identical mappings are used, and no cluster node network is favored over another.

The second type of switch model compares networks with different topologies. Performance curves for different types of lattices, trees, and De Bruijn networks are compared. In order to characterize the "goodness" of a mapping, a quality measure must be established. Changing the number of ports affects link speed and average hop count. Since the mapping algorithm has no impact on link speed, but does affect average hop count, the latter is an appropriate measure. In particular, as the number of ports is increased, the average hop count should decrease in a manner similar to that observed in the analytical studies. If it can be shown for each application program that the average hop count decreases "as it should", then it can be concluded that the mapping algorithm does not bias the results to favor (say) lattices with a small number of ports. On the other hand, if increasing the number of ports creates an unexpectedly small (large) improvement in average hop count, then a better mapping was done on the network with a small (large) number of ports, weakening (strengthening) the conclusion that a small number of ports is better.

Average hop count is defined for an application program  $\alpha$  as:

$$\bar{H}_\alpha = \frac{1}{\gamma} \sum_{i,j} \gamma_{ij} d_{ij}$$

where  $d_{ij}$  is the number of links traversed in the shortest path from  $i$  to  $j$ , and

$\gamma$  is the total number of messages sent into the network.  $\gamma_{ij}$  is the total number of messages sent from  $i$  to  $j$ . This definition differs from that presented earlier because the earlier definition assumed a uniform traffic distribution, implying all paths have equal weight. Values for  $\bar{H}_g$  are given in table 3.2 for the different application programs.

**Table 3.2.**  
**Average Hop Count**

		Lattices			De Bruijn		
		p=3	p=4	p=6	p=4	p=6	p=10
<b>BARNWELL</b>	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.850	2.575 9.65%	1.982 31.18%	2.387	1.875 21.45%	1.587 33.51%
<b>BLOCK I/O</b>	Lb. (l)	1.870	1.652 11.86%	1.392 25.56%	1.739	1.585 10.01%	1.392 19.95%
	map 1	2.620	2.471 5.89%	2.092 20.15%	2.944	2.081 29.31%	1.627 44.74%
	map 2	3.827	3.827 0.00%	3.479 9.09%			
<b>BLOCK STATE</b>	Lb. (l)	1.333	1.222 8.33%	1.111 16.85%	1.278	1.222 4.38%	1.111 13.07%
	map 1	3.259	3.000 7.95%	2.314 29.00%	2.630	2.000 23.95%	1.370 47.91%
	map 2	2.649	2.482 6.30%	2.316 12.57%			
<b>FFT</b>	map 1	5.542	4.792 13.53%	3.625 34.59%			
	map 2	2.867	2.187 18.75%	1.917 28.12%			
<b>LU</b>	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.979	2.585 13.23%	2.044 31.39%	2.515	1.983 21.15%	1.638 34.87%
<b>RANDOM</b>	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.692	2.483 7.76%	1.911 29.01%	2.459	1.925 21.72%	1.556 36.72%

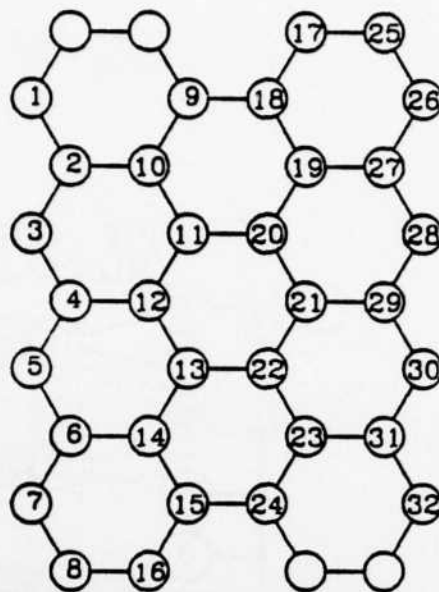
l.b. (l) = lower bound from weighted average hop count, optimal packing

ant. (g) = anticipated from 20 node network, global communications

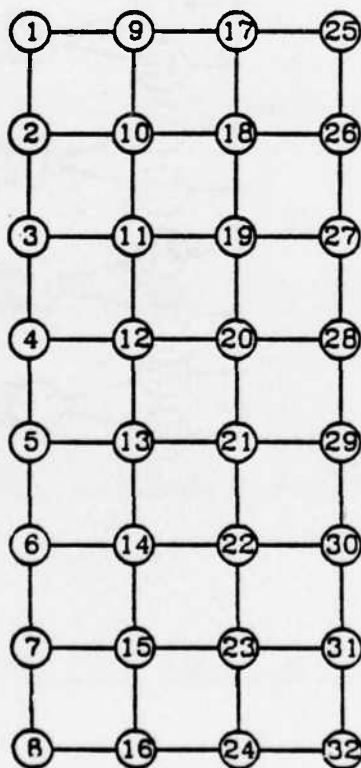
Two mapping schemes were used in the application programs exhibiting local communications (Block I/O, Block State, and FFT). The first is the original mapping whose results were described in section 3.7. This mapping was designed to minimize the average hop count on virtual circuits carrying the initial data samples in the SISO programs. If the processors of the network are envisioned as being uniformly distributed across the surface of a disk, the processor distributing the data samples resides in the center, and the processors receiving these samples are packed around it.

The second mapping attempts to optimize the virtual circuits carrying the local, i.e. pipelined, communication traffic among the computation tasks. This mapping was performed for lattice topologies only. Figures 3.22a-e indicate which tasks are assigned to which processors for this latter mapping. The FFT program uses the same task number assignments that are shown in figure 3.6. Figures 3.23a-f are the performance curves for these mappings for lattices with communication components and lattices with processor and communications integrated onto the same chip. Except for the FFT program, which will be discussed later, the results are qualitatively the same as those found in the original mapping.

Also included in table 3.2 are "anticipated" values for  $\bar{H}_0$ . For programs using global communications (Barnwell, LU, and Random), this anticipated value is computed by examining the average hop count in a 20 node network, assuming a uniform traffic distribution. For programs exhibiting local communications however (Block I/O, Block State, and FFT), this is clearly an unrealistic measure. Here, a lower bound value for anticipated hop count is used. Suppose the application program requires that processor  $X$  must send messages to  $k$  other processors. The minimum hop count from  $X$  to these processors is obtained by assuming that the  $k$  processors communicated with are those which are closest



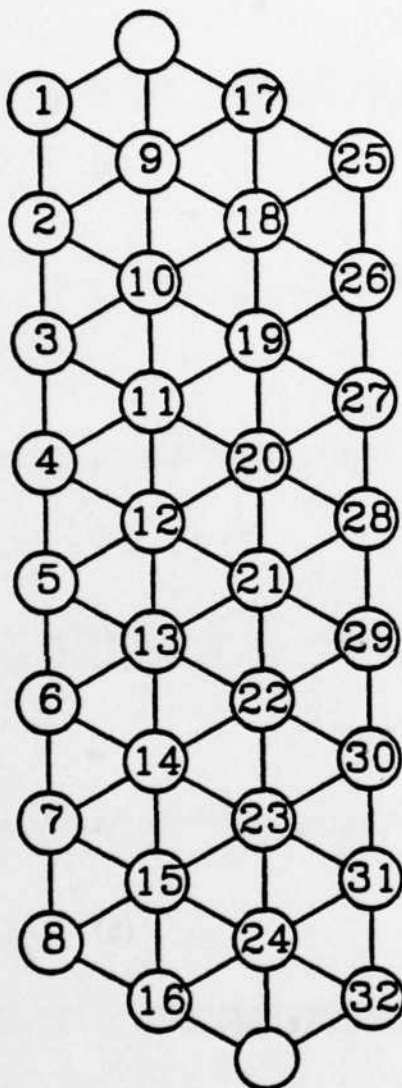
(a)



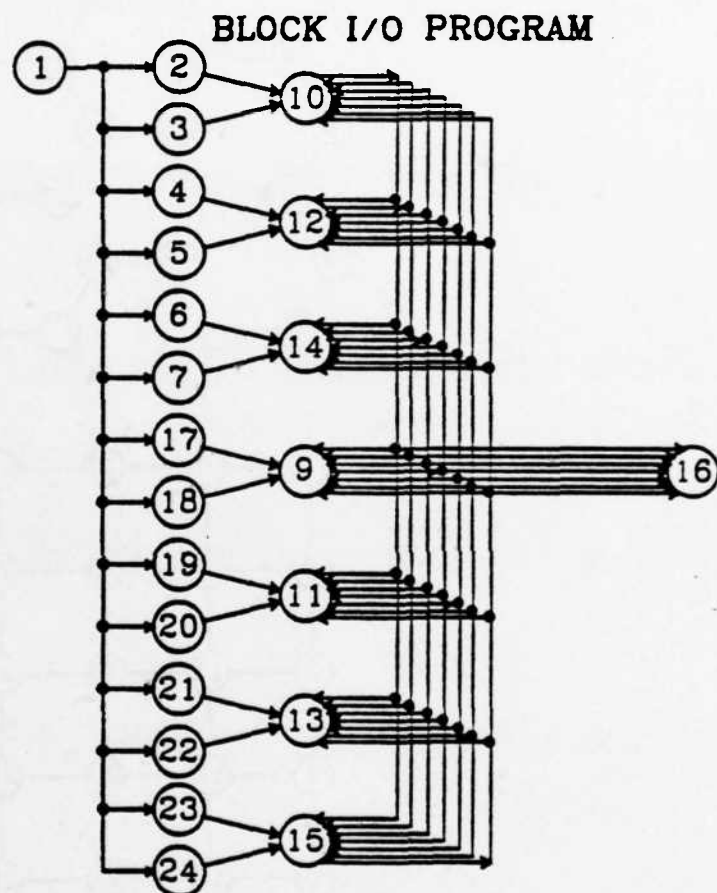
(b)

**Figure 3.22.** *Second mapping, lattices (a) 3 ports. (b) 4 ports.*





(c)



(d)

**Figure 3.22.** *Second Mapping (c) 6 ports. (d) Block I/O program.*

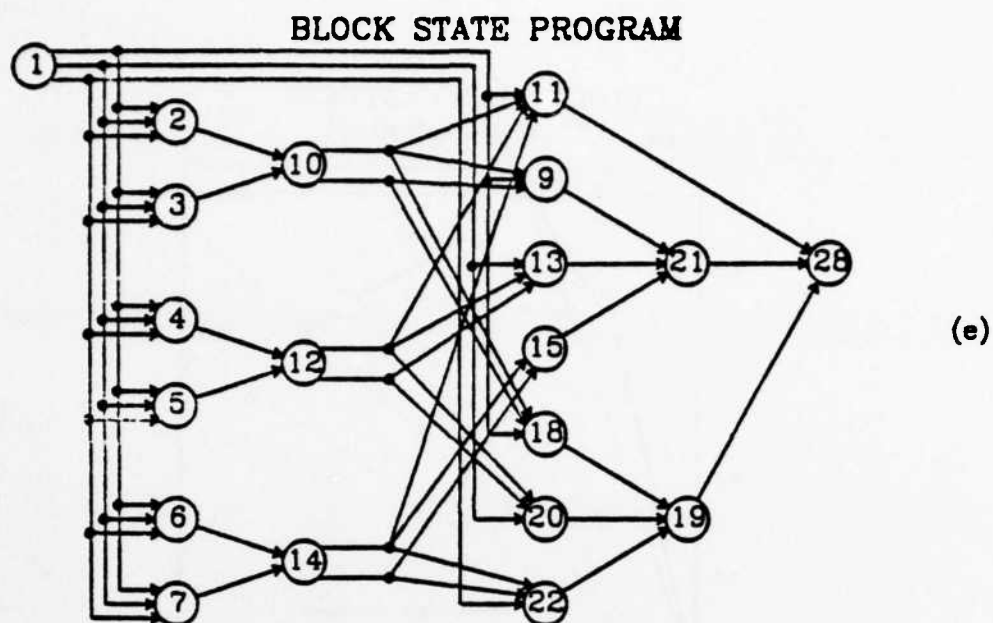


Figure 3.22. Second Mapping (e) Block state program.

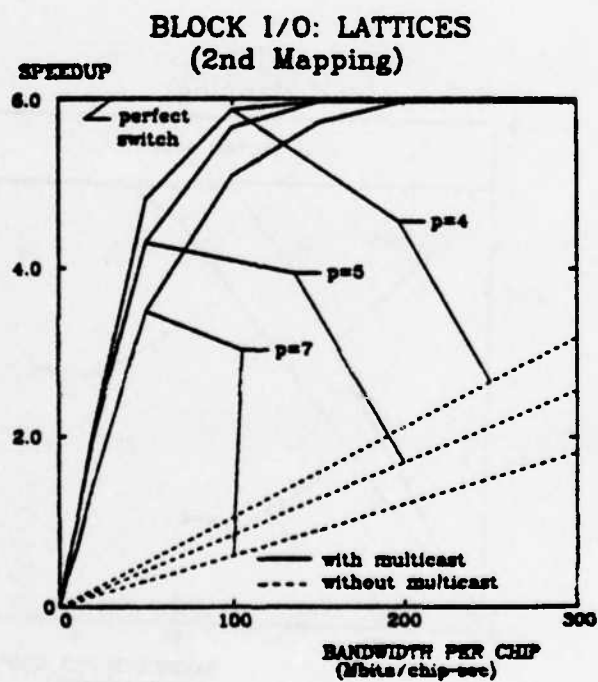


Figure 3.23. Lattices (2nd mapping) (a) Block I/O.

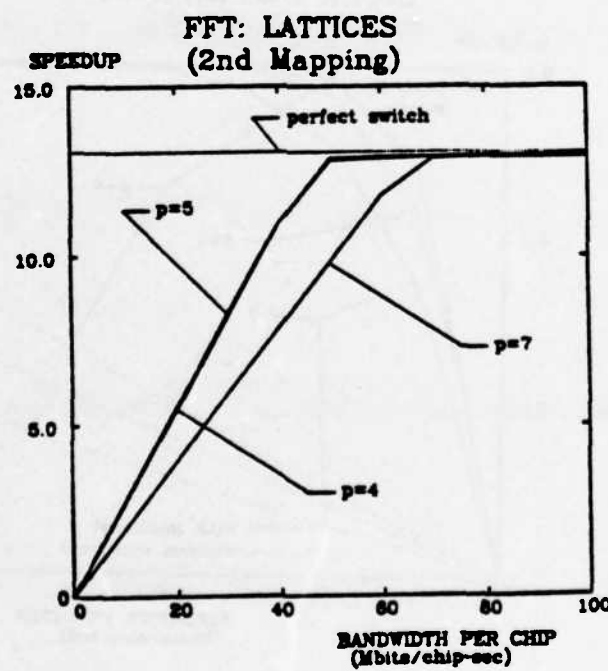
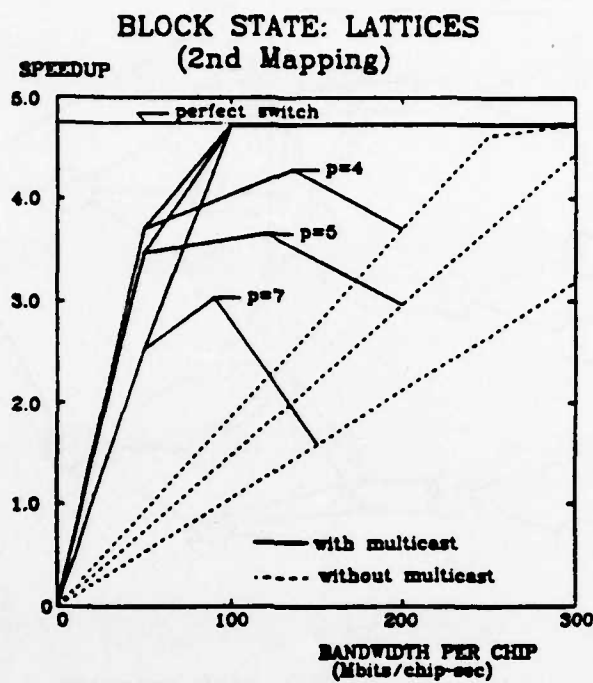


Figure 3.23. Lattices (2nd mapping) (b) Block State. (c) FFT.

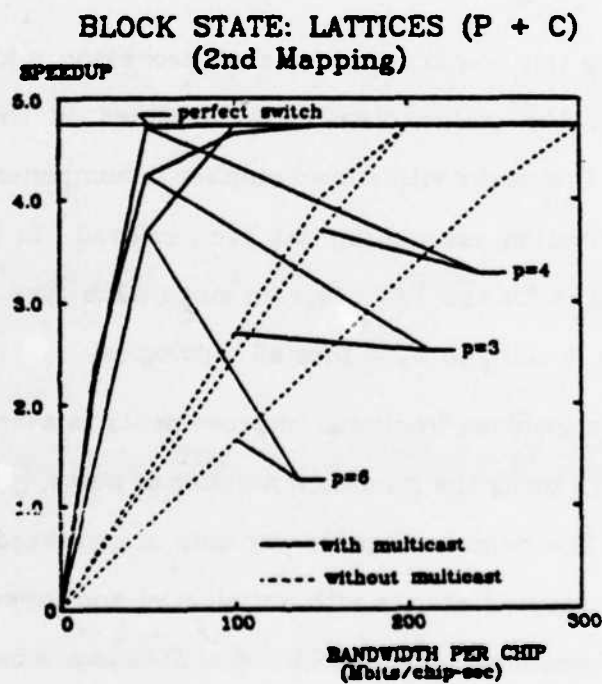
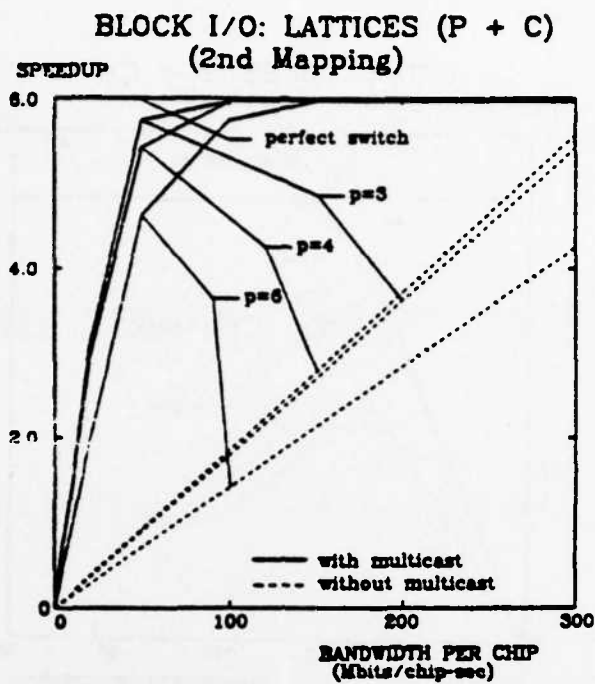
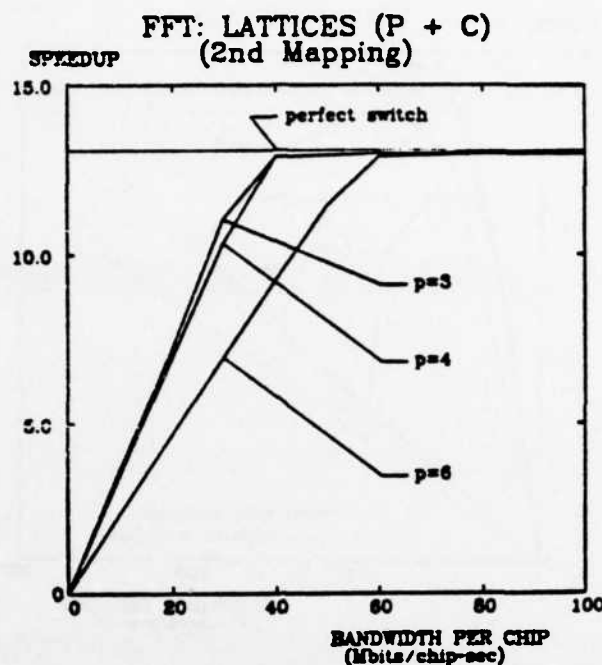


Figure 3.23. Lattices (2nd mapping, P+C) (d) Block I/O. (e) Block State.



(f)

Figure 3.23. *Lattices (2nd mapping, P+C) (f) FFT.*

to  $X$ . By repeating this computation for each processor, a lower bound on  $\bar{H}_s$  is obtained. This is the traffic distribution assumed in the analytical model presented earlier (networks with a fixed number of components), except the uniform traffic distribution assumption has been relaxed. In table 3.2, no anticipated value is listed for the FFT program since each task communicates with only 2 other tasks, leading to  $\bar{H}_s = 1$  for all topologies.

Table 3.2 also includes fractional improvements in average hop count relative to the network using the minimum number of ports, ( $p=3$  for lattices,  $p=4$  for De Bruijn) as the number of ports per chip is increased. In comparing the experimentally measured results with anticipated and lower-bound hop counts, it is seen that in most cases the experimental results are comparable to or surpass the anticipated improvement. This implies that any bias introduced by differences in the mapping algorithms makes a large number of ports appear in a better light than they should, thus strengthening the conclusion that a small number of ports is better.

Finally, let us consider the impact of using a better mapping algorithm on the results derived so far. As a closer match is found between the communication structure of the application program and the network topology, communications become more localized. Processors communicate less with processors far away, so the reduction in average path length, which occurs when the number of ports is increased, become less effective. The strength of the argument for a large number of ports relies on a reduction in network congestion. However as the mapping is improved, congestion becomes less significant. Increasing the number of ports only decreases the bandwidth available to each virtual circuit, and thus degrades performance. Thus, the fact that the simulation results may not use an "optimal" mapping of tasks to processors can only bias the results to favor a large number of ports, strengthening the conclusion that a small number of ports is better. Some support for this conclusion is seen in figure 3.23f, where an optimized mapping for the FFT program yields better performance with a small number of ports, while the original mapping yielded better performance with a large number of ports. Similarly, as shown in figure 3.12c, the execution of the FFT on the butterfly topology, an optimal mapping, yields better performance when a small number of ports is used.

### 3.9. Precision of the Simulations Results

A certain amount of uncertainty exists in all of the simulations presented thus far. The arrival times of messages at each node is a function of the queuing delays encountered in previous nodes, which in turn depends on the arrival times of other messages. These complex interactions lead to message delays which vary according to the times at which messages are generated. In general, the application programs executed on the multicomputer are not known a priori, so some uncertainty exists in the times at which messages are generated by the application program, leading to uncertainty in message delays. If this



uncertainty is large, the results presented thus far could be based on chance behavior rather than the tradeoffs described earlier. The amount of this uncertainty will now be examined.

The fact that the programs generated a relatively large number of messages (typically, thousands) combined with the large quantity of curves producing the same qualitative results leads one to suspect that the conclusions presented thus far are *not* simply the result of random fluctuations. Furthermore, the fact that the curves yielded results which were consistent with those predicted by the analytical models (or in disagreement in explainable ways) strengthens this belief. Nevertheless, in order to obtain a quantitative measure of the uncertainty described above, simulations were repeated with fluctuations introduced in the times at which messages are generated.

The artificial traffic generator program was executed on the hexagonal lattice network of figure 3.14a for the case of processor and communication circuitry integrated onto the same chip. This network was chosen because it does not exhibit any of the bottlenecks described earlier, e.g. the root bottleneck in the tree network, which might mask the effect of the random fluctuations. Interarrival times are again selected from an exponential distribution. The results of these experiments are shown in figure 3.24. Each curve represents performance for a different set of message arrival times. Fluctuations are introduced by using different seeds in the random number generator which determines interarrival times. As shown in figure 3.24, the delay in the lightly loaded network varies by up to 1.7%, while network throughput varies by up to 8.8%. Uncertainty in delay does increase significantly as the network approaches saturation, however this does not affect the conclusions derived above since they were based on the previous two performance measures.



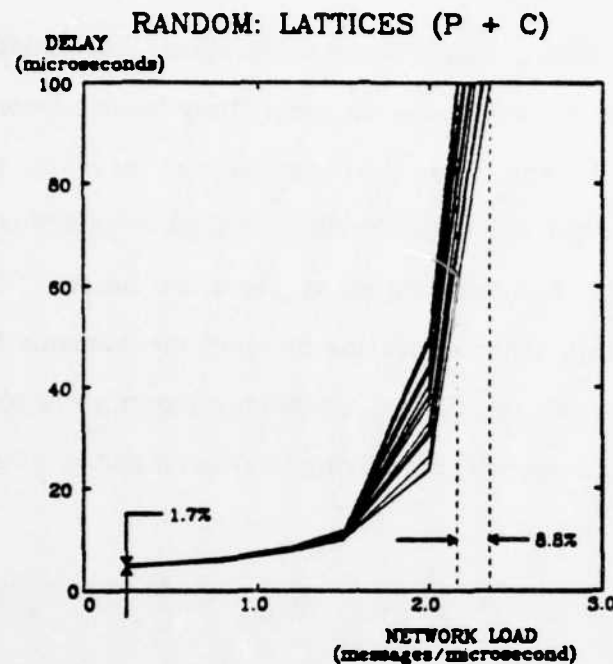


Figure 3.24. *Precision of simulations.*

Message delay uncertainty leads to uncertainty in the execution time of the programs. This latter quantity is the performance measure used in the bulk of the simulations presented here. The percentage of uncertainty in execution time will be *less* than that corresponding to message delay however, because message delay only affects one component of the overall execution time. Execution time is composed of two components, the time spent executing instructions and the time spent waiting for data. Message delays only affect the latter component. Although the absolute magnitude of fluctuations may be the same for both message delay and execution time, the percentage of the uncertainty will be smaller in the latter, since it is always larger. Thus, the uncertainty in message delay described above will lead to an even smaller uncertainty in execution time.

### 3.10. Summary of Simulation Studies

In most cases, the simulation results support the analytical results discussed earlier. When discrepancies do occur, they favor networks using a small number of ports. It was seen that bottlenecks are the source of these discrepancies. Thus, the simulation results support the conclusion found earlier that components with a small number of ports are better. These results also demonstrate the utility of incorporating efficient mechanisms for handling multiple destination messages. Such a mechanism can yield significant performance improvement in algorithms relying heavily on global information.

## CHAPTER FOUR

### DESIGN AND IMPLEMENTATION OF COMMUNICATION COMPONENTS

This chapter examines the amount of circuitry required to implement a VLSI communication component. Alternative mechanisms for transporting data through communication networks are first compared, and a *virtual-circuit* transport mechanism is argued to be the most attractive alternative for the networks discussed here. Details of such a transport mechanism are then described. Next, alternative schemes for providing hardware support for three key communication functions: routing, buffer management, and flow control, are described. Practical figures for the number of channels and buffers within each component are derived and used as the basis for estimates of the complexity of such a component. It will be seen that the functional capabilities of VLSI chips are now sufficient to allow the construction of communication components with enough buffer space and virtual channels to provide high-bandwidth communications in multicomputer networks.

#### 4.1. Transport Mechanisms

Since the primary function of the communication network is to move data, a transport mechanism, i.e. the means by which data is transmitted through the network, must be selected. A classification tree which includes the various transport mechanisms in use today is shown in figure 4.1. A number of characteristics which distinguish these transport mechanisms are also shown. Briefly, these characteristics are:

- (1) *Data Unit*: The unit of data transported through the network is either a variable-length message or a fixed-length packet.

- (2) *Routing Overhead*: The overhead associated with message routing is incurred either on a hop-by-hop basis at each node in the network or only in the initial set-up of a circuit.
- (3) *Bandwidth Allocation*: Bandwidth is allocated by the network either statically, e.g. when a circuit is set up, or dynamically as messages enter the network.
- (4) *Buffering Complexity*: The complexity of the buffering hardware varies with the sophistication of the chosen transport mechanism.

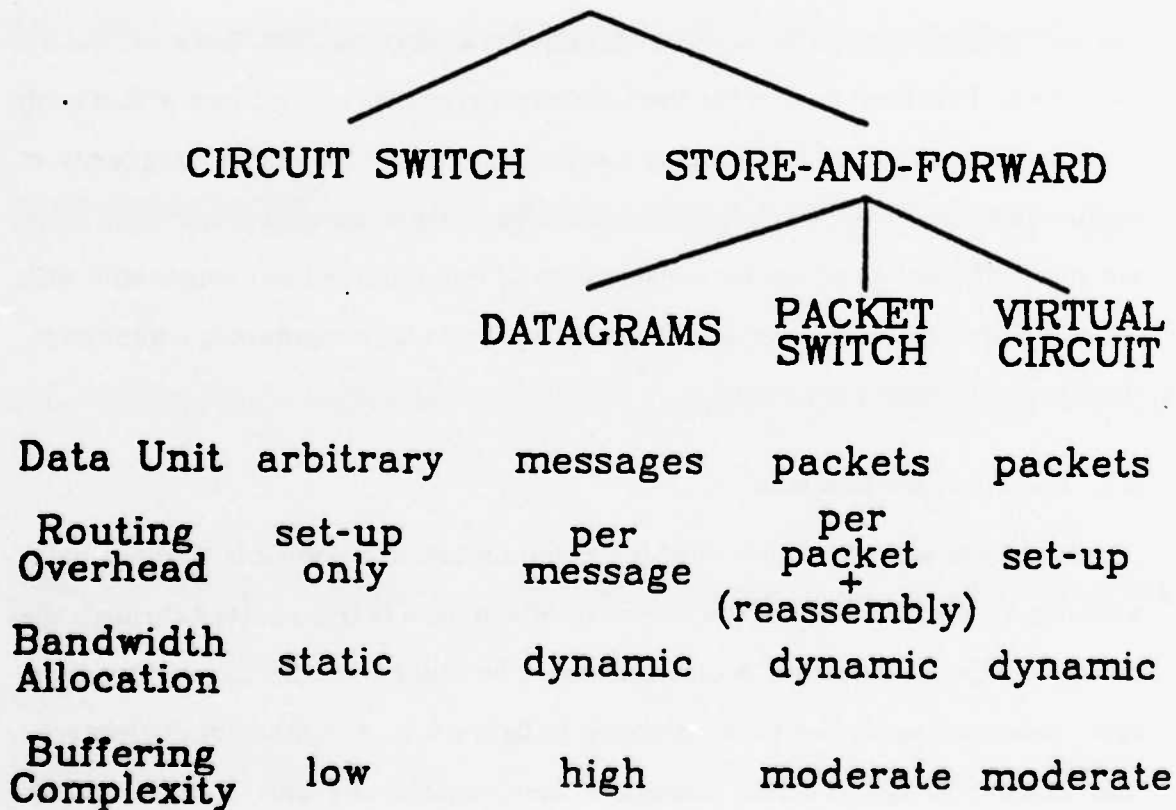


Figure 4.1. Transport Mechanisms.

According to the tree in figure 4.1, the first alternative to consider in selecting a transport mechanism is between a circuit-switched and a store-and-forward approach. The circuit-switched approach is best exemplified (at least conceptually) by the telephone system: When someone picks up a telephone and dials a phone number, a circuit is established between the caller and the party being called. Once this circuit is established, it remains intact until either party hangs up. Examples of circuit-switched networks are described in [Joel79, Mass79]. The most distinguishing characteristic of this approach is the fact that communicating parties are guaranteed a certain bandwidth and maximum latency when the call is established. Since the communication network cannot know when data will be transmitted, bandwidth must be allocated statically when the call is set up. Otherwise, the bandwidth may not be available when it is needed. Users are allocated a certain amount of bandwidth regardless of whether or not they actually use it. If communications are bursty, as is often the case in computer networks, much of the network's bandwidth will be wasted. This is the primary disadvantage of the circuit-switched approach.

However, the circuit-switched approach also offers a number of advantages. Routing overhead is usually paid only when the circuit is set up, so subsequent messages can flow through the network with little delay. This reduces the average delay on circuits carrying more than one message. Also, buffering strategies are simpler than those required for store and forward networks because bandwidth allocation is performed statically. If circuit-switching is used, the network can be designed to ensure that the rate of traffic flow into each node of the network never exceeds the rate of flow out, alleviating buffer overflow problems. In fact, if each circuit is implemented as a physical electrical connection between the communicating parties (e.g. a series of relays), the network need not provide any buffering at all!

Store-and-forward networks avoid the wasted bandwidth problem described above by allocating bandwidth dynamically to messages as they enter the network. Three types of store-and-forward networks have been implemented in the past:

- (1) Datagram networks.
- (2) Packet switched networks.
- (3) Virtual circuit networks.

Datagram networks are characterized by the unit of data sent through the network — variable length messages. Since each message can be relatively large, communication components would have to provide a relatively large amount of buffer space to hold arriving messages. This implies that a large amount of circuitry in each component must be devoted to messages buffers. In addition, since messages may vary in length, variable size buffers must be used. This increases the complexity of the buffer management circuitry significantly, since the buffer selected for a particular message must be at least as large as the message. This problem is identical to the difference between virtual memory systems based on segments, whose complexity usually requires a software implementation, and those based on pages, which are usually implemented, at least to a large extent, in hardware. Finally, routing overhead in the datagram approach is worse than that of the circuit-switched approach because routing decisions must be made on a hop-by-hop basis with each message sent into the network.

The packet switched transport mechanism alleviates many of the buffering problems described above. Here, each message is divided into a number of (usually) fixed-sized packets which are routed separately through the network. An end-to-end scheme is required to reassemble the message from its constituent packets. Since packets can be relatively small, buffering requirements in

each component are reduced. The use of fixed-sized packets also simplifies the buffer management circuitry. This approach does incur a significant amount of overhead on an end-to-end level to reassemble messages however. Since packets are routed separately through the network, they may follow different routes to the destination node, and therefore may arrive in an order different from that at which they were sent. The other disadvantage of the packet switched approach is that the routing overhead problem is worse than that of the datagram scheme, since this overhead now occurs on every *packet* rather than on every *message*.

If we examine the transport mechanisms described thus far, we see that the circuit-switched approach suffers from static bandwidth allocation, while the packet switch approach suffers from reassembly and routing overhead. One might hope that a hybrid which combines these two approaches can achieve the best of both mechanisms without their respective disadvantages. This is the motivation behind the *virtual circuit transport mechanism*, which is a mixture of packet switched and circuit-switched techniques. Here, a *virtual circuit* is established between processors which wish to communicate. A virtual circuit is a fixed, unidirectional path through the network from one processor to another. All messages sent on this circuit travel along this path to reach their destination.

Let us consider the characteristics of the virtual circuit transport mechanism (listed in figure 4.1). Like the packet switched mechanism, the data unit is a fixed sized packet (simplifying the buffering problems of the datagram mechanism). Routing is similar to the circuit-switched approach to the extent that the routing algorithm need only be applied when the circuit is set up, and not with subsequent packets. It will be seen however, that some overhead is still required to route messages, so the routing overhead is intermediate between



the circuit switched and the datagram/packet switched approaches. Since a store-and-forward mechanism is used, network bandwidth is allocated dynamically, although allocation is not as adaptive as it is in packet switched networks because packets are constrained to follow a fixed path from source to destination. The fixed path restriction in the virtual circuit mechanism is necessary to reduce routing overhead and to avoid reassembly overhead. Thus, while packet switched networks may be able to achieve higher bandwidth along an end-to-end connection by utilizing multiple paths between the two processors, the virtual circuit scheme will yield lower latency on individual messages since they spend less time in each node waiting for routing decisions to be made. In addition, the virtual circuit mechanism can utilize multiple paths between two nodes by establishing several circuits between the two processors.

Thus, a virtual circuit transport mechanism appears to be the most attractive for the networks described here. Details of the operation of this mechanism are described in the next section. Hardware implementations are described in the sections which follow.

## **4.2. A Virtual Circuit Based Communication System**

The communication domain studied here is a packet-based network using a virtual circuit transport mechanism. Mechanisms for establishing, maintaining, and tearing down circuits are described in this section.

### **4.2.1. Virtual Circuits**

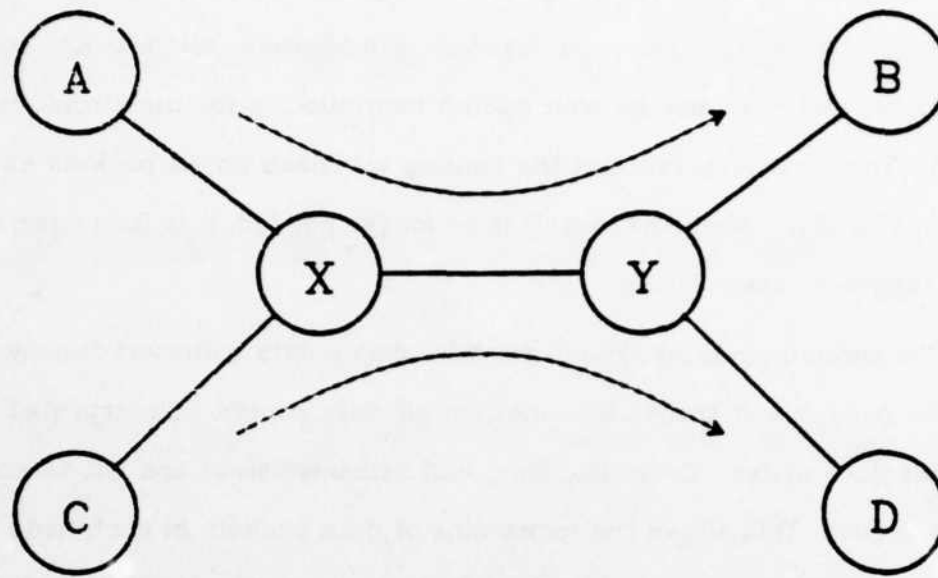
Each processor has a fixed number of input and output circuits for receiving and sending data respectively. Sending a message is a three step process. First, a virtual circuit (i.e. a path of time-multiplexed links) to the destination processor is established by sending a message header with routing information through the network. Once a circuit is set up, an arbitrary amount of data,

which may consist of several logical messages, can be sent along this circuit. Data can follow the message header immediately without an end-to-end handshake and need not be transmitted continuously for the circuit to remain intact. This approach reduces the routing overhead on all packets except the message header. When the circuit is no longer needed, it is torn down by sending a tagged message trailer.

The communications system provides only a data transport facility. Except for the header and trailer information, all data passes uninterpreted through intermediate nodes. Error checking and retransmission are left to an end-to-end protocol. This allows the forwarding of data packets in each node to begin before the entire packet has arrived if the proper outgoing link is idle (virtual cut-through [Kerm79]). If error checking and retransmissions were performed within the network on a hop-by-hop basis, forwarding could not begin until the entire packet has arrived and was checked for errors, since otherwise an erroneous packet would have been forwarded by the time the error was detected. This end-to-end approach is justified by the low error rates observed in local computer networks [Shoc80]. Since the networks discussed here cover an even smaller geographic area, and thus are less susceptible to environmental noise, this assumption is even more appropriate.

#### 4.2.2. Virtual Channels

The communications domain can be viewed as a simple, connected graph. Nodes and edges represent communications components and links, respectively. A circuit from one processor to another corresponds to a path in this graph. Two distinct paths (say from node A to B and from C to D in figure 4.2) may use a common edge (from X to Y). Thus, the link associated with that edge must be multiplexed between the two paths, and provisions must be made to ensure that data from A is sent to B, and not to D.



**Figure 4.2.** *Two Paths Multiplexed through the Same Link.*

Each physical link is divided into some fixed number of unidirectional *virtual channels*. Each channel can carry data for one virtual circuit (i.e. one path). Thus, a circuit from one node to another consists of a sequence of channels on the links in the path between the two nodes. The circuit from A to B in figure 4.2, for example, might use channel #3 to get to X, then #5 to get to Y, and finally #7 to get to B.

When node X sends data to node Y, the latter must determine which circuit this data belongs to. Two commonly used techniques for providing this information are, among others:

- (1) Divide the link into a fixed number of *time slots* and statically assign each time slot to a channel (e.g. the first time slot might be assigned to channel #0, the second to channel #1, etc.). The time slot on which the data arrives identifies the channel that sent it.

- (2) Precede the data with a tag that identifies the channel it is being sent on.

In this scheme, the available bandwidth on the link is allocated to the various channels by some demand-driven scheduling algorithm.

In the first scheme, the link is effectively divided into a number of lower bandwidth links, with the sum of these bandwidths equal to that of the physical link. If a channel does not send any data, its allocated bandwidth is wasted. In addition, latency is increased since each channel must wait for its turn to send a unit of data. In the second scheme, the entire bandwidth of the link can be allocated to channels upon demand, i.e. when they have data to send, so the inefficiencies associated with the previous approach are avoided. However, some bandwidth is required to carry the channel tag. Demand-driven time-multiplexing is superior if the degree of multiplexing on each link is high, and many channels do not always have data to send. This is often the case in computer-to-computer communications, so the dynamic approach is more suitable for the networks described here.

#### 4.2.3. Routing Hardware

In order to route messages through each node of the network, channels entering a node (input channels) must be "linked" to channels leaving the node (output channels). Each node maintains a set of *translation tables* to perform this function. There is one translation table for each input port of a node. Each entry of the translation table contains two fields: an output port, and the number of a channel on that port. When data arrives on an input channel, say channel #3, entry 3 of the translation table for that port is read to yield the output port and the number of the channel the data is to be forwarded on.

The translation tables logically link incoming and outgoing channels, and thus establish the various virtual circuits through the node. Setting up these circuits involves allocating channels and updating translation tables along each

path from source to destination. This task is performed by a *routing controller* residing in each communications node.

Initially, all translation tables specify that data is to be sent to the local routing controller. When a message header setting up a new circuit arrives at a node, the routing controller analyzes the destination address in the header, determines the proper output port with the use of some routing algorithm, allocates a free output channel, and updates the translation table at the input port. Measurements on a TTL prototype of such a routing controller [Fuji80] show that this entire operation can be done in 4-5  $\mu$ sec if a free channel is available on the selected link. Subsequent data is then forwarded without intervention by the routing controller. Similarly, when the circuit is torn down, the channel is released, and the corresponding translation table entry is reset to point to the routing controller.

#### 4.2.4. Packet Types and Formats

Three types of packets have been discussed thus far: a "set-up packet" which establishes virtual circuits, a "trailer packet" which tears them down, and a "data packet" which carries data. In addition, it is useful to provide a "clear packet" which flows through a virtual circuit, removing any data packets it encounters along the way. Such a mechanism is useful in error recovery protocols to reset virtual circuits to a "known" state.

Packets must be tagged to distinguish the various types. Each packet is preceded by a header which indicates the packet's type, as well as a channel number indicating which virtual circuit the packet belongs to. Set-up packets also carry routing information (e.g. a destination address) which the routing controller uses to set up the circuit. Assuming two bits to indicate type, one bit for parity, and a one byte header on each packet (excluding routing information on set-up packets), 5 bits remain for a virtual channel number. This implies a

maximum of 32 channels can be supported on each link. Later, it is argued that under current technology, a larger number of channels should be supported, say 64 or 128. Since it is convenient to restrict the header information to an integral number of bytes, a two-byte header could be used to support this many channels.

An alternative approach to the fixed length header scheme described above is to use variable length packet headers. For example, assuming that most of the packets flowing through the network are data packets, we could confine the overhead in these packets to a single byte, while forcing other packet types to use several bytes. Under this scheme, the header of each data packet consists of a 7-bit channel number and a single bit for parity. One channel number, say #0, is declared to be "undefined". When a packet header specifies this channel number, it indicates that the packet is not a data packet, but rather some other type. Subsequent bytes indicate the type of packet, and any type-specific information. This approach reduces the overhead required on data packets, and thus provides better performance in transmitting these packets than the fixed-length header scheme described above. Although the amount of time required to process the other packet types, the set-up packet in particular, is slightly increased, delays on these other packet types are less crucial. The assumption that data packets use a single byte for header information was used in much of the analysis presented earlier.

#### **4.3. Key Functions of the Communication Component**

Any communication component must provide mechanisms for routing messages to their proper destination, managing the limited amount of buffer space, and controlling the rate at which packets flow from one node to another. Hardware implementation of these mechanisms is required to achieve high-performance. Mechanisms to perform these functions are outlined in this



section. Hardware implementations are described in the section that follows. Implementation of other portions of the communication component, i.e. the I/O ports and routing controller, are only briefly summarized since they are described elsewhere [Laur79, Wong81, Fuji80].

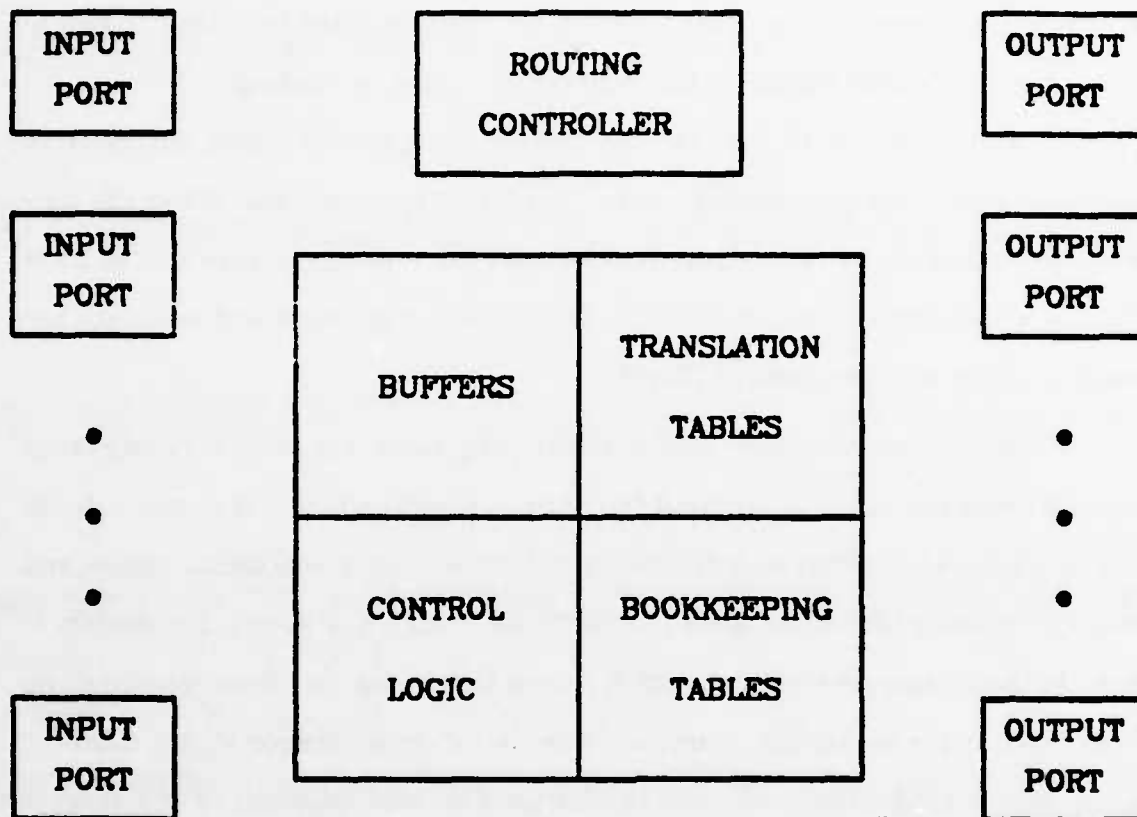
A block diagram indicating the functions that must be provided by each component is shown in figure 4.3. The component contains three or more ports, each accommodating a link to a neighboring node. It also contains a certain amount of buffer memory, bookkeeping tables, and control logic. Translation tables logically link incoming and outgoing channels, and thus establish the various virtual circuits through the component. Finally, a microcoded engine called the routing control is responsible for setting up virtual circuits and implementing less frequently used network functions such as failure recovery protocols.

#### 4.3.1. Routing

All communication networks require some routing algorithm to build the paths, i.e. the virtual circuits, between nodes sending and receiving messages. A great deal of research has been done in the area of routing in loosely coupled computer networks, and much of this work is applicable here [Gerl81, Tane81]. In the context of the proposed communication domain, we will only consider totally distributed routing that does not rely on a centralized authority. For this discussion it is also appropriate to distinguish between regular networks with a predefined topology, such as arrays or binary trees, and irregular networks of arbitrary connectivity.

In regular networks, routing can be performed in each node by a state machine which performs a fixed algorithm based on the local and destination addresses. In square lattices, for example, the routing controller could forward the message header in a direction that would reduce the difference between the x- or y- coordinates of the current and the destination nodes. Routing





**Figure 4.3.** *Functions provided by communication component.*

algorithms for binary half-ring and full-ring trees have been discussed elsewhere [Sequ78].

For a general-purpose communication component, the routing algorithm must not be frozen in hardware. A routing controller with a writable program memory is more appropriate and guarantees that the same component can serve many different network topologies. A routing algorithm suitable for the particular network structure could be broadcast at system initialization.

For irregular networks, routing may be based on suitable lookup tables. In a decentralized system each node  $i$  has entries of the form:

$$NN = R_i(DN).$$

implying that messages destined for node  $DN$  are forwarded by node  $i$  to neighbor node  $NN$ . This lookup table, commonly called a *routing table*, can be defined statically, or it can be maintained dynamically using information exchanged between neighboring nodes. The latter approach also allows the network to automatically reconfigure itself should the topology change due to node failure or network expansion [Taji77]. Techniques to initialize and maintain the routing tables are discussed in [Gerl81].

If the network has many nodes, the routing table will be excessively large since a separate entry is required for each destination node. A common technique which reduces the size of this table is to employ hierarchical names and multiple routing tables per node [Kamo76]. An example of such a mechanism is seen in the telephone system in which names (telephone numbers) consist of an area code and a seven digit number. When a call to a number with a different area code is made, the area code is first used to route the call to the correct area, and then the phone number is used to locate the final destination. Conceptually, routing could thus be performed as follows:

- (1) If the area code of the destination matches that of the router, then the seven digit number is used to locate the next node via a "neighborhood" routing table.
- (2) If the area code does not match that of the node doing the routing, then the area code is used to look up the next node via an "area code" routing table.

The remaining seven digits in the phone number are ignored.

Thus, a two-level naming hierarchy is used along with a routing table for each level. Such a scheme reduces the table size by grouping nodes which are far away into a single entry in the "area code" routing table.

One can easily extend this principle to an arbitrary number of naming levels. To determine the number of levels required to minimize the storage space required for routing tables, let there be  $l$  levels, with  $g_i$  entries in the level  $i$  table. The object is to minimize  $g_1 + g_2 + \dots + g_l$  subject to constant  $N = g_1 \times g_2 \times \dots \times g_l$ , the number of nodes in the network. It is easy to show that this sum is minimized for

$$g_1 = g_2 = \dots = g_l = e \quad \text{and} \quad l = \ln N,$$

where  $e$  is approximately 2.718. Thus, to minimize the table size in each node, there should be many levels with few entries in each level [McQu74].

The reduction in table size resulting from a multi-level routing scheme can be substantial. A 16-bit destination address partitioned into eight 2-bit fields requires eight 4-entry routing tables, or a total of 32 entries. The single-level routing table would require 65,536 entries. The routing controller described in [Fuji80] uses a single-level lookup table with 256 entries. A hardware implementation of a hierarchical routing scheme will be presented later.

#### 4.3.2. Buffer Management

Each message passed into the communication domain must be subdivided by the sender into some number of fixed-length packets. As discussed earlier, allowing variable length packets adds a considerable amount of complexity to the component. These packets form the unit of data transmitted across the links of the communication domain. Due to conflicts that arise when several packets simultaneously require the use of the same link, buffering is required in each node. The communication component must have some strategy for managing these buffers.

A scheme is necessary to allocate a node's buffers among the virtual circuits using the node. A simple solution is to give each channel on each link a

separate buffer. This is inefficient however, since much of the buffer space will be unused most of the time. By allowing several channels to share buffers, fluctuations in the need for buffer space can be averaged over a large number of communication paths, and fewer buffers are required to achieve the same performance. A mapping is then required to link each channel to the buffers holding packets for that channel so that they can be found when it is time to forward them. Furthermore, when a new packet arrives, an empty buffer must be found. From this perspective, buffer management is similar to the management of a cache memory: a program (here, a channel) must fit blocks from main memory (packets) into cache pages (buffers).

As in cache memory design, there are three well known schemes for performing this mapping:

- (1) direct mapping
- (2) set-associative mapping
- (3) fully associative mapping

In turn, these three schemes offer an increased degree of buffer sharing, and thus improved memory utilization, but at the cost of increased complexity in the control circuitry. They are distinguished by restrictions on where a channel's packets can be placed. In the direct mapping scheme (minimal sharing), each channel has a set of buffers dedicated to it, i.e. its own fifo buffer queue. The set-associative scheme (moderate sharing) allows each channel to use a larger set of buffers, but it is no longer given sole access to them. This scheme might be implemented by letting all channels of a single port share a pool of buffers dedicated to this port. In the fully associative scheme (maximal sharing), each node has a centralized pool of buffers which all channels share. Implementations of the set-associative and fully associative schemes will be discussed in later sections. An implementation of the direct mapping scheme has

been described previously [Laur79, Sequ78].

#### 4.3.3. Flow Control

Flow control refers to the mechanism which regulates the transmission of data packets along virtual circuits. The network must be able to "throttle" traffic on virtual circuits to prevent buffer overflow (such mechanisms are sometimes referred to as congestion control in the literature [Tane81]), and to handle situations in which a processor is sent more messages than it can immediately receive. In addition to providing a *mechanism* which allows components to throttle traffic, a *policy* is also required to determine which virtual circuits must be throttled, and when. Such a policy will be discussed next, followed by a discussion of different throttling mechanisms.

Since one of the purposes of flow control is to avoid buffer overflow, a natural policy is to begin throttling traffic when the pool of free (i.e. empty) buffers becomes depleted. If a node is inundated with data, packets will "back up" along the virtual circuits leading up to it, much like the way cars back up on a congested freeway. This type of flow control, called "back pressure flow control", is analogous to water (packets) flowing through a pipe (buffers). If the pipe becomes blocked or constricted, water backs up to its source. Such a mechanism has been used successfully in TYMNET, a loosely coupled, commercial communication network [Tyme81].

The flow control policy described above can lead to a problem called "buffer hogging". Here, one virtual circuit uses more than its share of the buffers in a node. If a virtual circuit becomes blocked, e.g. due to a congested output link, packets may continue to arrive on that virtual circuit and occupy most, or all of the buffers in the node. Without some mechanism to restrict buffer sharing, buffer hogging will impede other traffic using the node and lead to deadlock situations. This situation can be avoided by controlling the maximum number of

buffers each channel can use. It might be noted that the direct mapping scheme, and to a lesser extent the set-associative scheme, automatically provide some protection against buffer hogging, since they inherently restrict buffer sharing. All three schemes however, need some mechanism to ensure that data is not lost if no free buffers are available.

Thus, in order to prevent buffer hogging, each output channel may not hold more than some "channel limit" of buffers at once. Even with this restriction however, another form of buffer hogging may still arise. A congested output link could use all of the node's buffers and block traffic on other links. To prevent this, each output port is restricted to using no more than some maximum number of buffers, determined by a higher level protocol. This maximum number, called the "port limit", can be changed dynamically to shift additional buffers to highly utilized ports, while still providing some space for traffic on lightly loaded ports. Studies indicate that by restricting the number of buffers an output port can use "output port buffer hogging" is prevented, and a significant improvement in the bandwidth provided by the node is obtained [Irla78]. These studies also indicate that as a general rule, each port should not be allowed to use more than  $b/\sqrt{p}$  buffers in a  $p$ -port node with  $b$  buffers.

Assuming a buffer allocation policy is used to control the rate of packet forwarding, let us now examine the flow control mechanism itself, i.e. the mechanism which performs the actual throttling. Two mechanisms, sender-controlled and receiver-controlled throttling, will be discussed. They are characterized by whether the sending or the receiving node implements the policy described above. The receiver-controlled mechanism is the simpler mechanism, and will be described first.

The receiver-controlled flow control mechanism can be implemented by a send/acknowledge protocol to transmit data over the link. In this scheme, each

node sends a packet, and waits for the receiver to return a control signal indicating whether it accepted or rejected (i.e. discarded) the packet. An "ack" signal denotes an accepted packet while a "nack" denotes a rejected packet. If a nack is returned, the packet must be retransmitted at a later time.

A receiver may choose to reject a packet because of buffer space limitations or transmissions errors. Here, it is assumed that communication components only check header information for transmission errors, since the virtual cut-through mechanism prevents retransmission if errors in the data are detected. With virtual cut-through, the first bytes of the packet may have been forwarded to the next node before an error in later bytes is detected, making immediate recovery difficult, if not impossible. Errors in data bytes must be handled by an end-to-end protocol which detects and retransmits damaged packets.

It is also assumed here that each link has a separate control line to carry the ack/nack signal back to the sender. Alternatively, the control signal could be piggy-backed onto a packet going in the opposite direction, however, this leads to a "looser coupling" between sender and receiver, forcing the sender to either deal with multiple unacknowledged packets pending over the link [Pouz78] (adding a considerable amount of complexity to the circuitry in the port), or to stop using the link until the acknowledgement arrives (wasting bandwidth). Since the receiver can generate an acknowledgement after only the header is received, and since a direct connection to the sender is available for transmitting this signal, this scheme offers the unusual feature that the sender will receive the acknowledgement before it has finished sending the packet! This allows a virtual circuit to "pipeline" a stream of packets through an otherwise idle node without incurring the delays associated with waiting for acknowledgements or the complexity of multiple unacknowledged packets.



An alternative approach to flow control is to implement the buffer allocation policy for a node in its neighboring nodes, i.e. control the flow of information from the sender rather than the receiver end of each link. For example, each output port could maintain a table remembering how many buffers in the neighboring node are allocated to each channel of the link connecting the two. With this information, the sender can decide which channel to serve next, and packets can be forwarded without the risk of overflowing the buffer space in the receiver. Maintaining this remote status information requires some overhead: The fact that the receiver has freed up a buffer must be reported back to the transmitter. Finally, since packets cannot be retransmitted, transmission errors in packet headers result in lost packets. An end-to-end mechanism is required to retransmit these packets.

As in the send/acknowledge flow control scheme, buffer hogging is prevented by controlling the number of buffers used by each channel. It might be noted however, that output port buffer hogging is much more difficult to prevent. This is because the size of the queue on an output link depends on the packets received from the node's neighbors. When these neighbors send packets, they do not know which output port in the receiving node the packet will use, since routing decisions are made inside the receiver. Thus the neighbors cannot control the queue size on a specific link, and nothing prevents a single port from monopolizing the entire buffer pool.

The send/acknowledge protocol leads to a simple implementation, while the remote buffer management approach prevents rejected packets, and thus avoids retransmissions and waste of bandwidth. Both schemes require some overhead to provide the feedback signals necessary for flow control. In the send/acknowledge scheme, dedicated pins are used, while in the remote buffer management scheme, piggy-backed control signals are required. Implementa-

tion and comparisons of these two mechanisms will be described in the sections which follow.

#### 4.4. Implementation of VLSI Communication Components

Hardware implementations of the communication functions described above are outlined in this section, and two designs are presented which integrate these functions into a single chip. The first is a Y-component design using a set-associative buffer management scheme and remote buffer allocation for flow control. It will be seen that there are a number of severe deficiencies in this design. The second design, which corrects these deficiencies, uses a fully associative buffer management scheme. Implementations of both sender-controlled and receiver-controlled flow control mechanisms are also discussed. Common to both designs is the routing controller with hardware support for hierarchical routing. This is the subject of the next section. The two designs are described in subsequent sections.

##### 4.4.1. Routing Hardware

This section describes a hardware implementation of the hierarchical routing table mechanism described earlier. This hardware is part of the routing controller which is responsible for setting up virtual circuits through the node. The remainder of the routing controller is described in [Fuji80].

When a virtual circuit is being constructed, the routing hardware is given a hierarchical destination address, and must determine which output port the virtual circuit is to use. This is accomplished by a set of routing tables, one for each level of the hierarchy, as discussed earlier. The routing controller, part of which is implemented as a microprogrammed engine, is responsible for loading and maintaining the routing tables, e.g. by a shortest path routing algorithm.

Two implementations are discussed. The first assumes that routing tables at all levels are the same size, some power of 2. The second design relaxes this assumption, but at the cost of added complexity.

An  $l$ -level hierarchical node address consists of a string of digits,  $A_{l-1}A_{l-2} \cdots A_1A_0$ . Digit  $A_i$  is used to index the routing table at level  $i$ . A routing table entry contains either an output port number indicating which port to use, or a "NULL" flag indicating that the table on the next level must be searched. Let  $RT_i$  denote the routing table at level  $i$ , with  $0 \leq i < l$ . The algorithm to determine the appropriate output port is as follows:

```

level := 0;      /* current level, 0, 1, ... l-1 */
while (( $RT_{level}[A_{level}] = \text{NULL}$ ) and (level < l))
    level := level + 1;
if (level < l)
    return ( $RT_{level}[A_{level}]$ );    /* return output port */
else
    return (NULL);    /* destination node reached */

```

If this routine returns NULL, then the message has reached its final destination, i.e. the destination address matches the local address. Otherwise, the number of the output port selected by the routing algorithm is returned.

One hardware implementation of this table lookup mechanism is shown in figure 4.4a. It is assumed that each table contains  $2^k$  entries. The bus widths in figure 4.4a assume that there are 8 levels, and 4 routing table entries in each level (i.e.  $k=2$ ). The "address register" holds the destination address. The rightmost  $k$  bits of this register hold  $A_0$ , the next  $k$  bits hold  $A_1$ , etc. A single RAM holds all of the routing tables. The upper bits of the address lines of this RAM specify a routing table (i.e. a level), and the lower bits specify an offset into this table. The lower  $k$  bits of the address register (i.e.  $A_{level}$  in the program

above) are concatenated with the output of the level counter (the current level) to form this address. The shifter aligns the destination address bits by shifting out  $A_k$  and moving  $A_{k+1}$  into the rightmost position each clock cycle. Since each bit is shifted exactly  $k$  bits on each clock, the shifter can be implemented by an edge triggered register and a simple permutation of wires. Finally, not shown is the control logic which sequences through the various routing tables. Design of this finite state machine is straight-forward, using the level counter and circuitry to detect NULL routing table entries, and generating signals to shift the address bits and increment the level counter.

A second implementation, shown in figure 4.4b, relaxes the "fixed routing table size" restriction. The bus widths shown in this figure support up to 8 levels and a total of 256 routing table entries. The number of levels and sizes of the various routing tables is programmable at system initialization. The "address RAM" holds the base addresses of the various routing tables. Entry  $i$  contains the base address of the routing table at level  $i$ . The routing tables are again stored in a single RAM. The routing table offset,  $A_k$ , is generated by masking appropriate bits of the address register. This offset is added to the base address to generate an address for the routing table RAM. A barrel shifter aligns data in the address register for the next iteration. The mask bits and the number of address bits to be shifted are stored in the "mask RAM" and "shift RAM" respectively. These RAMs are loaded by the routing controller at initialization. Together, their contents describe the format of the address register. The control logic for this second implementation is virtually the same as that of the previous design.

#### 4.4.2. A Y-Component Design

The design of a Y-component has been studied [Wong81]. Details of this design will be repeated here as an example of one implementation of the

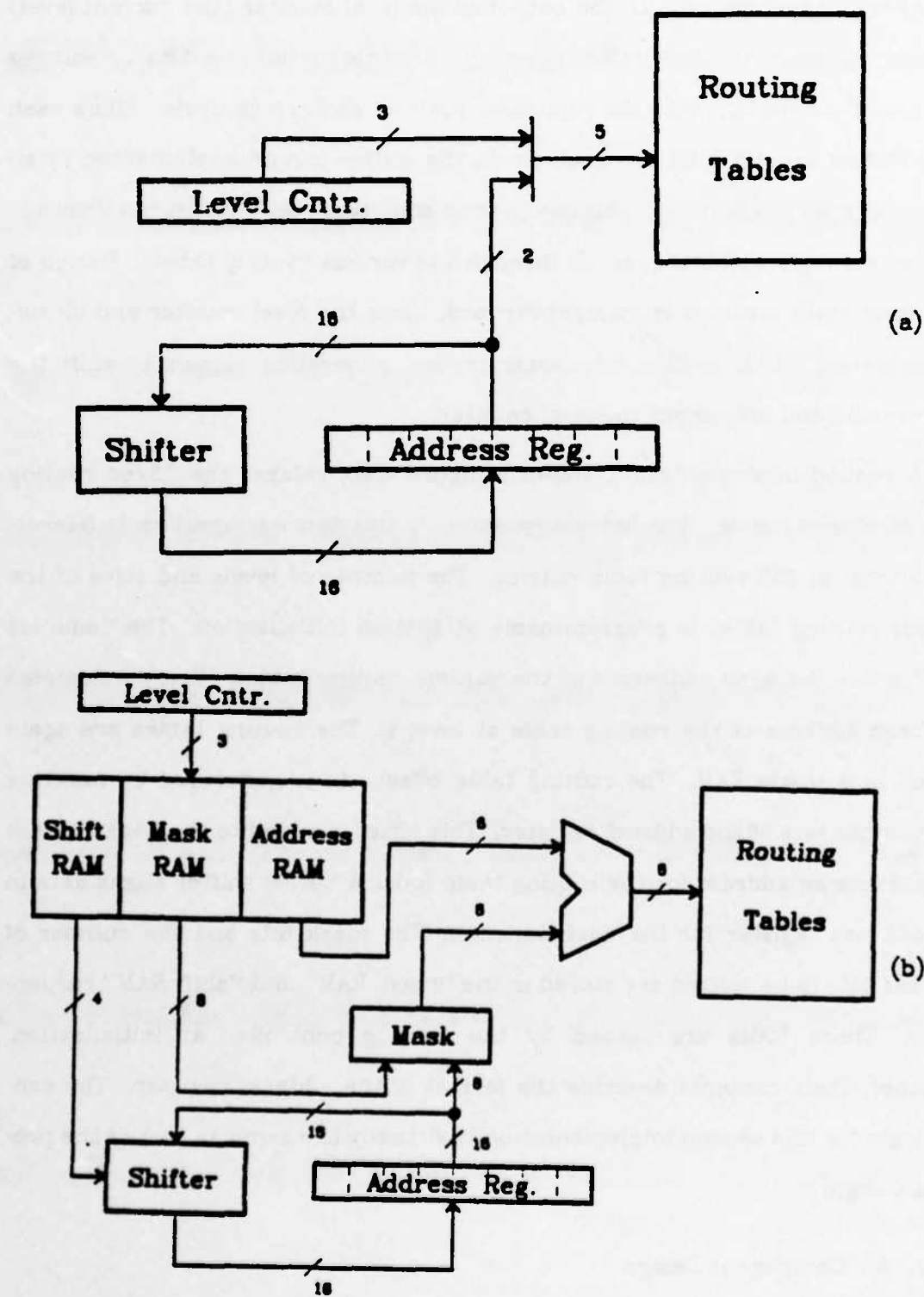
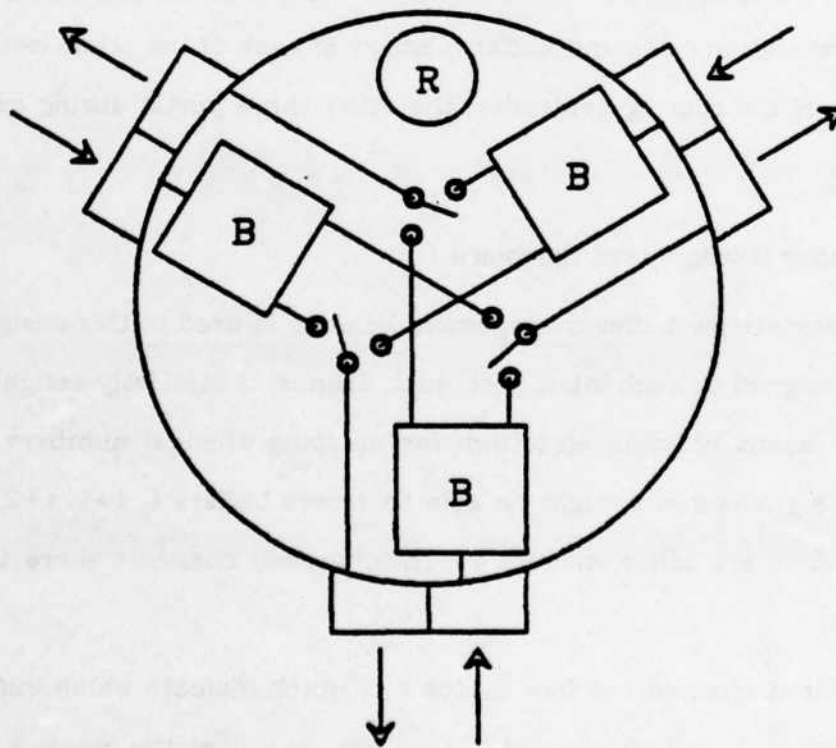


Figure 4.4. Hierarchical routing circuitry (a) simple. (b) complex.

functions described above. A block diagram for this design is shown in figure 4.5. The component consists of a routing controller (R), three input ports, three output ports, and three buffer modules (B), one associated with each input port. It will be assumed that there are  $c$  input and  $c$  output channels on each port, and that each buffer module consists of  $b$  data buffers.

When a packet arrives at an input port, it is placed in one of that port's buffers. The routing controller (which can, for the moment, be considered a fourth output port) and the other two output ports actively search this input port's translation table and buffers to locate packets destined for it. When such a packet is found, it is forwarded and the buffer is marked empty. Some addi-



**Figure 4.5.** *Block diagram of Y-component.*



tional control logic ensures that packets on each channel are forwarded in the order in which they arrived.

Although the translation table and the buffer memory of a single input port can both be read at the same time, it is not possible to simultaneously perform two reads of the *same* translation table or buffer memory. To avoid conflicts, each "major clock cycle" (the time interval to transmit or receive a single word of data over the link) is subdivided into 4 "minor clock cycles", and these minor cycles are statically assigned to output ports to time multiplex access to the buffers and translation tables without contention. It is assumed that each translation table and buffer memory can be accessed during a single minor clock cycle. This assignment ensures that each output port has an opportunity to read the translation table and buffer memory of each of the other two ports (or in the case of the routing controller, the other three ports) during each major clock cycle.

#### 4.4.2.1. Buffer Management Hardware

A set-associative buffer management scheme is used in this design. Of the  $b$  buffers assigned to each input port, each channel is statically assigned (say) 4 buffers by means of some algorithm for mapping channel numbers to buffer addresses, e.g. channel  $i$  might be able to access buffers  $i$ ,  $i+1$ ,  $i+2$ , and  $i+3$ , where all sums are taken modulo  $b$ . Thus, several channels share the use of each buffer.

Each input channel has four status bits which indicate which buffers actually hold a packet for that channel. These bits, as well as the input port's translation tables are scanned by the other two output ports and routing controller to locate packets which must be forwarded.



#### 4.4.2.2. Flow Control Hardware

A remote buffer management scheme is used for flow control. The buffers of each input port are managed by the neighbor on the sender side of the link. In other words, the output port of each node is responsible for allocating buffer space in the neighboring node to the packets it sends.

In addition to the input port status bits described earlier, each output port maintains a bit map indicating which of the buffers of the input port on the other side of the link are free, and which are in use. When a component sends a packet, it not only specifies the number of the channel the packet is being sent on, but also the buffer that the receiving component is to use. It also must set the appropriate bit of the bit map to signify that the remote buffer is now in use. The input port receiving the packet then loads it into the designated buffer, and sets the appropriate input port status bit for the channel the packet arrived on, indicating that it has a packet waiting to be forwarded. When the appropriate output port sees that this bit has been set, it forwards the packet. The neighbor which originally sent it must be notified that this buffer is now free. A control byte piggy-backed onto a packet going to this neighbor accomplishes this task (a dummy packet is created if there is no traffic in this direction). Since the sender does not send a packet unless there is an empty buffer on the neighboring node to receive it, buffer overflow cannot occur.

#### 4.4.3. Deficiencies in the Y-Component Design

The design presented above suffers from a number of deficiencies. The most severe problem arises from the polling scheme used to determine which channels hold packets waiting to be forwarded: each output port polls the translation tables and the status bits of the other input ports. If there are  $c$  channels per port, then each port requires  $c$  major clock cycles to poll all of the input channels of the other two ports (two channels, one from each port, can be polled

in one clock cycle). If a packet arrives on an arbitrary channel, then an average of  $c/2$  clock cycles expire before that channel is polled. Later, it will be seen that  $c$  should be relatively large, say 128 or 256, so long delays result from this polling scheme. In addition, it will be seen that the number of buffers in each component need not be very large, say 16 or 32, so most channels do not have packets waiting to be forwarded. Many idle channels will have to be polled before a channel with data is found. Thus, channel polling is an unreasonably slow and inefficient mechanism to locate waiting packets.

The remote buffer management scheme described above wastes link bandwidth, since it requires more overhead than is actually necessary. In the previously described scheme, a buffer number precedes every packet sent over the link. This is required because the sender allocates buffers in the receiving node. The allocation function could be controlled by the receiver however, since the sender only needs to be sure that a remote buffer exists to hold each packet it sends, and does not need to know the address of the remote buffer. Thus, buffer numbers need not be transmitted over the link. A single counter indicating the number of free buffers in the remote node's input port could be used to provide the necessary information without incurring additional overhead on the link. Sending a packet decrements the counter, while receiving a signal indicating that the remote node forwarded the packet will increment it. The receiver is left the responsibility of determining which buffer each arriving packet should use. This approach eliminates the need to send buffer numbers over the link, and thus achieves more efficient use of the link's bandwidth. Details of such an approach will be described later.

The design described above requires several memory references to the same memory on each major clock cycle. For example, the translation table polling mechanism requires four memory references per clock. In addition, if

two output ports want to simultaneously forward packets from the same input port, two buffer memory reads per clock are required. The time required by these memory references could slow the clock rate, reducing the communication bandwidth of the entire network.

Finally, the studies which follow indicate that a high degree of buffer sharing is desirable, since there are many more channels than buffers. This increases the desirability of a fully associative buffer management scheme. One implementation of such a scheme is described next.

#### 4.4.4. An Alternative Design

In order to remedy the deficiencies described above, an alternative design for a communication component has been studied. Unlike the previous design, this design has been structured in such a manner that the number of I/O ports can be increased without adding unduly to its complexity. A fully associative buffer management scheme is explored, as well as two types of flow control mechanisms.

A block diagram of the communication component design is shown in figure 4.6. The most distinguishing feature of this design is a single pool of buffers shared by all channels of the component. Since all packets traveling through a node must use this pool, it must provide enough bandwidth to avoid becoming a bottleneck. This is achieved by interleaving the memory 16 ways, assuming packets consist of 16 bytes. Byte  $i$  of each packet ( $i \leq 0 < 16$ ) is always stored in memory module  $i$  ( $MM_i$ ). Each of the  $p$  ports can simultaneously load a packet into a buffer, provided no two use the same memory module at the same time. In the worst case,  $p$  packets simultaneously arrive at a node. Since only one port can be granted access to  $MM_0$ , additional registers are required to temporarily buffer the arriving data bytes until they can be stored in  $MM_0$ . On the next clock cycle, when the second byte of each packet arrives, one of these

newly arriving bytes will be loaded into  $MM_1$ , and one of the temporarily buffered bytes can now be written into  $MM_0$ . Similarly, three accesses to the buffer pool will occur on the third clock, and so on. Eventually, each port will be able to access a different memory module on each clock cycle.

If the links can transmit one data byte per clock cycle, then the communication component must be able to transport  $p$  bytes from the input ports to the memory modules in each clock. A high-speed, time-multiplexed bus performs this function. Since this bus remains entirely within the chip, it can run approx-

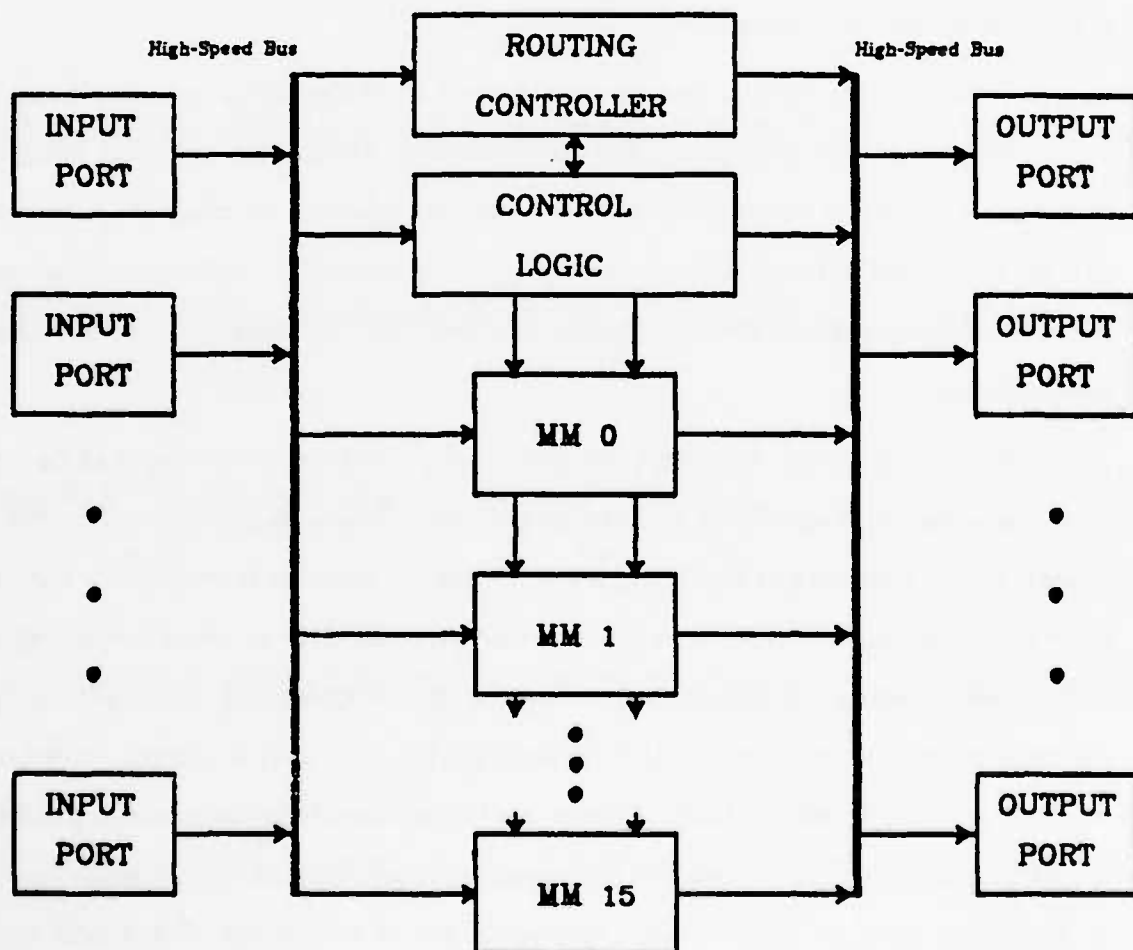


Figure 4.6. Block diagram of alternative design.

imately an order of magnitude faster than the I/O links, which require off-chip communications [Sequ78]. A second high speed bus carries bytes from the memory modules to the output ports. Single-port memories can be used in the memory modules provided the control logic only initiates one operation - forward a packet or receive a packet - per clock cycle. The designs which follow assume that this is the case.

A block diagram of the control logic module is shown in figure 4.7. Let us consider the events which occur when a packet arrives at the node. First, the header, i.e. the input channel number, arrives. The translation table is read to determine which output port and channel will be forwarding the packet. The output of the translation table is sent to both the buffer management and flow control modules. The buffer management module allocates an empty buffer to hold the newly arriving packet, and notes the location of this buffer as well as the output port/channel specified by the translation table. This information will be needed when it is time to forward the packet. The buffer module then sends the address of the buffer into  $MM_0$ , and the packet is stored, byte-by-byte, into successive memory modules on subsequent clock cycles. The flow control module notes that this output channel now has a packet waiting to be forwarded. When the output link specified by the translation table is free, the flow control logic sends a signal to the buffer management module indicating that the latter should forward the next packet waiting on this output channel. The buffer management module finds the address of the buffer holding this packet and sends it to  $MM_0$ . The packet is read from the buffer byte-by-byte, and forwarded over the output link. In both reading and writing a packet, the same memory address is pipelined from memory module to memory module on successive clock cycles. Since the pipelined structure of the memory modules allows the reading, i.e. forwarding, of a packet to begin before all of it arrives, virtual cut-

through is easily implemented.

The sections which follow give more detailed explanations of possible hardware implementations of these mechanisms. The next section describes two implementations of a fully associative buffer management scheme which differ in the number of buffers each virtual circuit can hold at one time. It is seen that a significant reduction in complexity is possible if this number is restricted to one. Following this, two possible implementations of flow control mechanisms are presented. The first uses a send/acknowledge protocol, while the second uses a remote buffer management scheme.

High Speed Bus

---

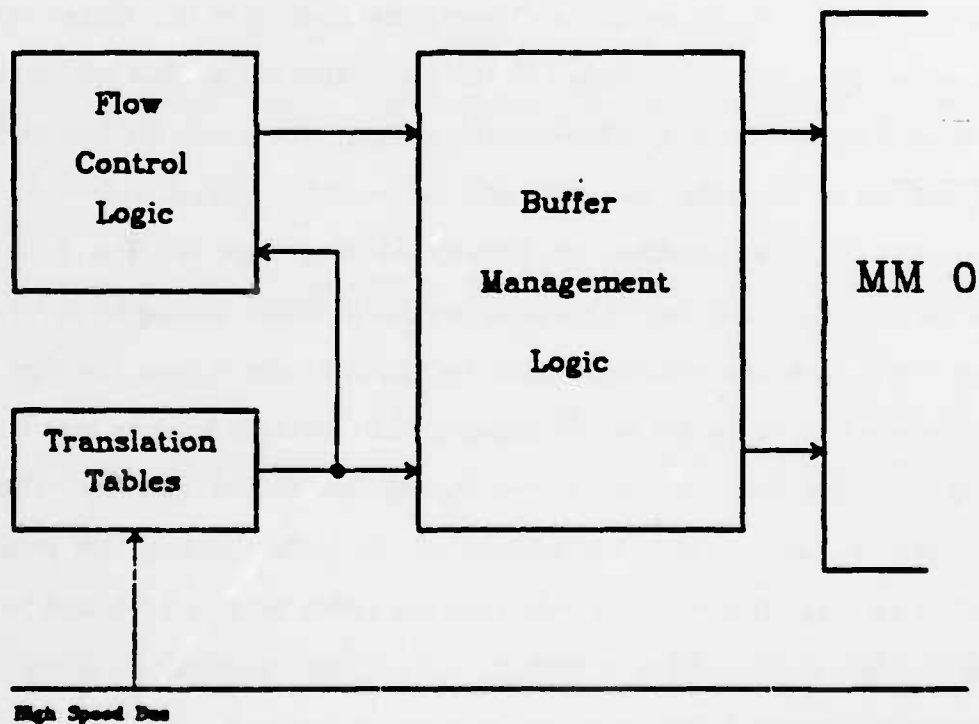


Figure 4.7. Control logic.

In the circuit diagrams which follow, the widths of data paths are based on a component with 4 I/O ports, 32 data buffers, and 128 channels per link. Thus, port numbers, buffer numbers, and channel numbers are 2, 5, and 7 bits in length respectively. These choices will be discussed later in this chapter.

#### 4.4.4.1. Buffer Management Hardware

The buffer management module must perform two functions:

- (1) Locate a free buffer to hold a newly arriving packet.
- (2) Locate the next packet waiting to be forwarded on a particular output channel.

Two implementations will be described for performing these functions. The first assumes that the number of packets waiting to use a given output channel can be larger than one. The second restricts this number to be at most one.

Since buffers are dynamically assigned to virtual channels on demand, a mechanism is required to keep track of which buffers are assigned to which channels at any given time. In the presented solution, this task is accomplished by "chaining" the buffers waiting to be forwarded on an output channel into a linked list for that channel. When a packet arrives, it is placed at the end of the linked list corresponding to the channel the packet is to be forwarded on (read from the translation table). It is removed from the list after it has been successfully transmitted to the next node. The linked lists are managed as a FIFO queue to ensure that packets are forwarded in the same order in which they arrived. The mechanisms for managing the linked lists are implemented in hardware so that packet forwarding can proceed as quickly as possible. A block diagram of one implementation is shown in figure 4.8a. In the discussion which follows, it is assumed that each  $p$ -port component has  $b$  buffers and  $c$  input (or output) channels per port, i.e.  $c \times p$  channels per node.



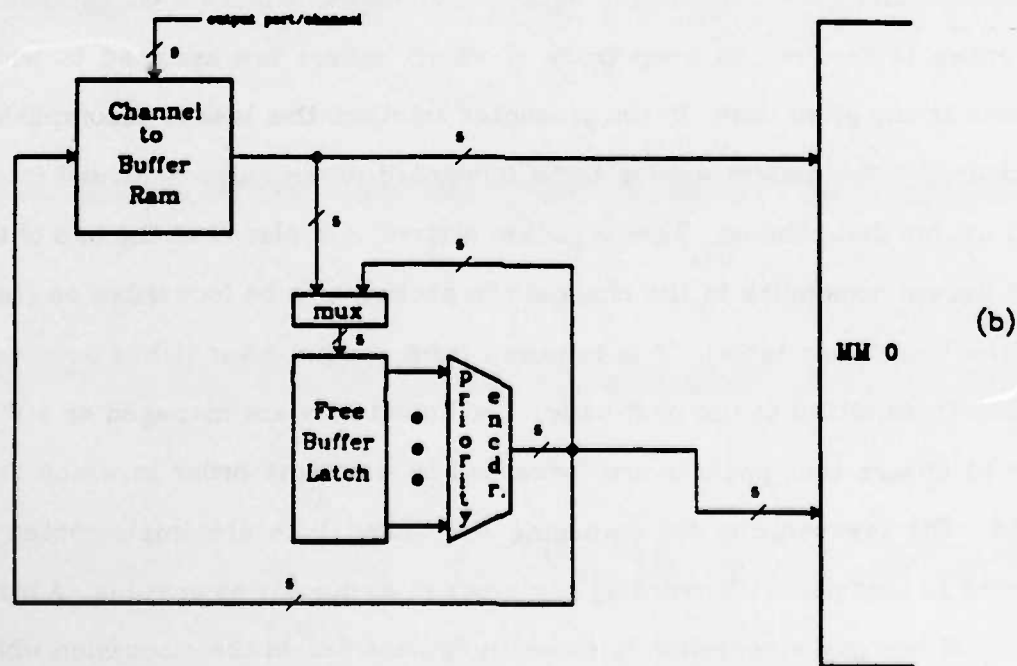
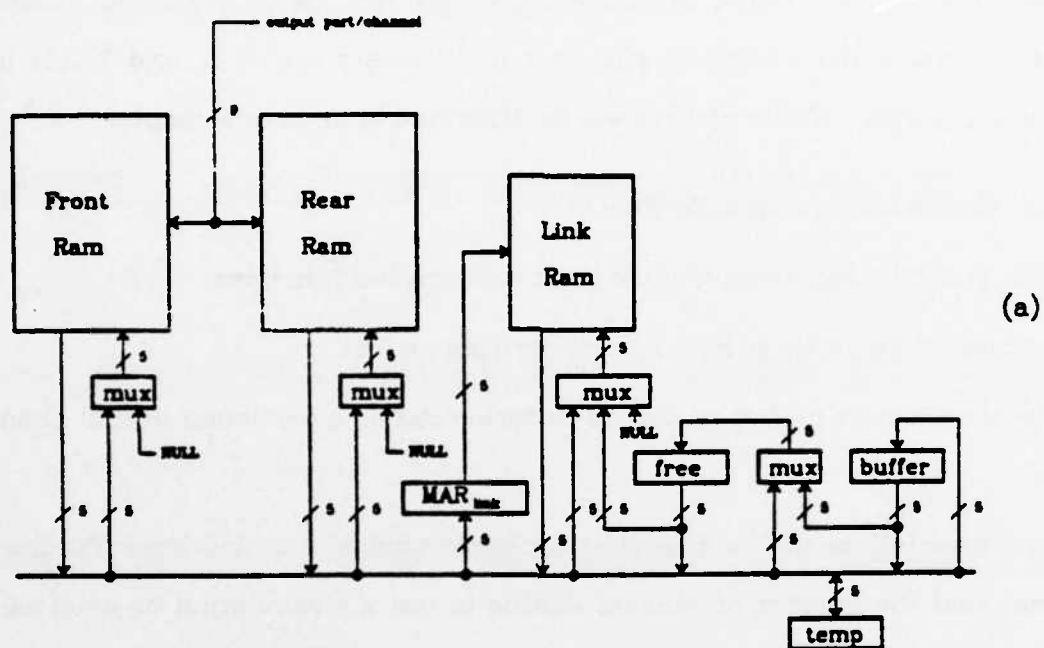


Figure 4.8. Buffer management circuitry (a)  $> 1$  buffers/channel. (b) 1 buffer/channel.

A buffer consists of a 16 byte data portion, which is physically distributed across the memory modules in figure 4.8, and a pointer word. The pointer word indicates the address of the next buffer in this buffer's linked list. The b-word "link" RAM in figure 4.8a holds these pointers. Each output channel has pointers to the buffers at the front and end of its linked list. The cxp-word "front" and "rear" RAMs in figure 4.8a perform this function. Adding a new buffer to an output channel implies reading the rear RAM (to find the last buffer in the list), and writing the address of the new buffer into this address of the link RAM (to set the new link) as well as the rear RAM (to set the pointer to the new rear element). Deleting an entry implies reading the front RAM (to get the address of the buffer being deleted), reading the link RAM (to get the new front element), and writing this latter address into the front RAM.

Buffers not linked to any channel list are empty, and are linked together in a separate "free list". A register, called the "free" register, points to the beginning of the free list. The arrival of a new packet implies removing an element, i.e. the address of a free buffer, from the free list, and adding this address to an output channel's linked list. Forwarding a packet implies removing the front element from the channel list, and adding it to the free list. Allowing simultaneous access to different memories, a buffer can be added to or deleted from a linked list in four and three clock cycles respectively (where each memory reference requires one clock cycle). The operations necessary to process an arriving/departing packet are shown in figure 4.9 below.

The complexity of the design described above can be reduced significantly if the restriction is made that any output channel can use at most *one* buffer at a time. The impact of this restriction on system performance will be discussed later. Most of the hardware for managing the linked lists can be eliminated, since the lists are at most one element long. This allows the three RAMs in figure

**Packet arrives on channel "ich":**

clock cycle	action	comments
1)	buffer $\leftarrow$ free; $MAR_{link} \leftarrow$ free; if (free = NULL) abort;	address of free buffer get ready to read new free list head no more free buffers
2)	free $\leftarrow$ Link[ $MAR_{link}$ ];	read new free list head
3)	Link[ $MAR_{link}$ ] $\leftarrow$ NULL; temp $\leftarrow$ Rear[ich]; $MAR_{link} \leftarrow$ Rear[ich];	mark pointer for new buffer locate end of linked list get ready to add to end of list
4)	Rear[ich] $\leftarrow$ buffer; if (temp = NULL) Front[ich] $\leftarrow$ buffer; else Link[ $MAR_{link}$ ] $\leftarrow$ buffer;	update pointer to end of list if channel list now empty then update front pointer else update previous last element

**Packet forwarded on channel "och":**

clock cycle	action	comments
1)	buffer $\leftarrow$ Front[och]; $MAR_{link} \leftarrow$ Front[och];	get address of first buffer in list
2)	if (buffer = NULL) abort; temp $\leftarrow$ Link[ $MAR_{link}$ ];	abort if list empty address of new front element
3)	Front[och] $\leftarrow$ temp; Link[ $MAR_{link}$ ] $\leftarrow$ free; free $\leftarrow$ buffer; if (temp = NULL) Rear[och] $\leftarrow$ NULL;	update front pointer add buffer to free list new front of free list check if list now empty

**Figure 4.9.** Operations to send and receive packets.

4.8a to be combined into one RAM, the "channel-to-buffer" RAM shown in figure 4.8b. This  $c \times p$ -word RAM maps output channels to buffer addresses. Word  $i$  holds the address of the buffer currently holding a packet for channel  $i$ . The list of free buffers is replaced by a  $b$ -bit latch, called the "free buffer latch". The free buffer latch is implemented as a bit-addressable latch, i.e. a memory device which is written as a RAM (one bit at a time), but read as a latch (all bits in parallel). Each bit indicates the status of a buffer: free (1) or in use (0).

When a new packet arrives, the buffer management circuitry must perform two operations, assuming the flow control circuitry has first established that the packet can be accepted (discussed later):

- (1) Find and allocate a free buffer.
- (2) Note the location of the buffer so that the packet can be found when it is time to forward it.

The address of a free buffer is determined by a priority encoder attached to the free buffer latch. The resulting address is sent to  $MM_0$ . This address is also used to clear the corresponding bit in the free buffer latch, effectively allocating the buffer, and completing the first operation. The second operation is accomplished by writing the address of the selected buffer number into the channel-to-buffer RAM at the memory location corresponding to the output channel responsible for forwarding the packet (read from the translation table).

Forwarding a packet on output channel  $i$  also requires two operations:

- (1) Locate the buffer holding the packet for channel  $i$ .
- (2) Release the buffer.

The first task is accomplished by reading address  $i$  of the channel-to-buffer RAM. The resulting address is used to set the corresponding bit in the free buffer latch, marking the buffer free to be used by other packets, thus accomplishing the second task.

Forwarding a packet requires the time of two memory operations since the channel-to-buffer RAM read must be completed before the latch write can be begun. These two steps are easily pipelined however, allowing a "send packet" operation to be initiated every clock cycle. The operations for receiving a packet can be performed in a single clock cycle since both can be executed concurrently. This is in contrast to the four clock cycles required in the previous buffer management scheme which used linked lists.

#### 4.4.4.2. Flow Control Hardware: Send/Acknowledge Protocol

In the send/acknowledge flow control scheme, each node sends a packet, and receives an acknowledgement signal indicating whether the receiver accepted or rejected (i.e. discarded) the packet. Packets may be rejected because of buffer space limitations or transmissions errors in the header and must be retransmitted at some later time. As discussed earlier, it is assumed that each link uses a separate control line to carry the acknowledgement signal back to the sender with minimal delay.

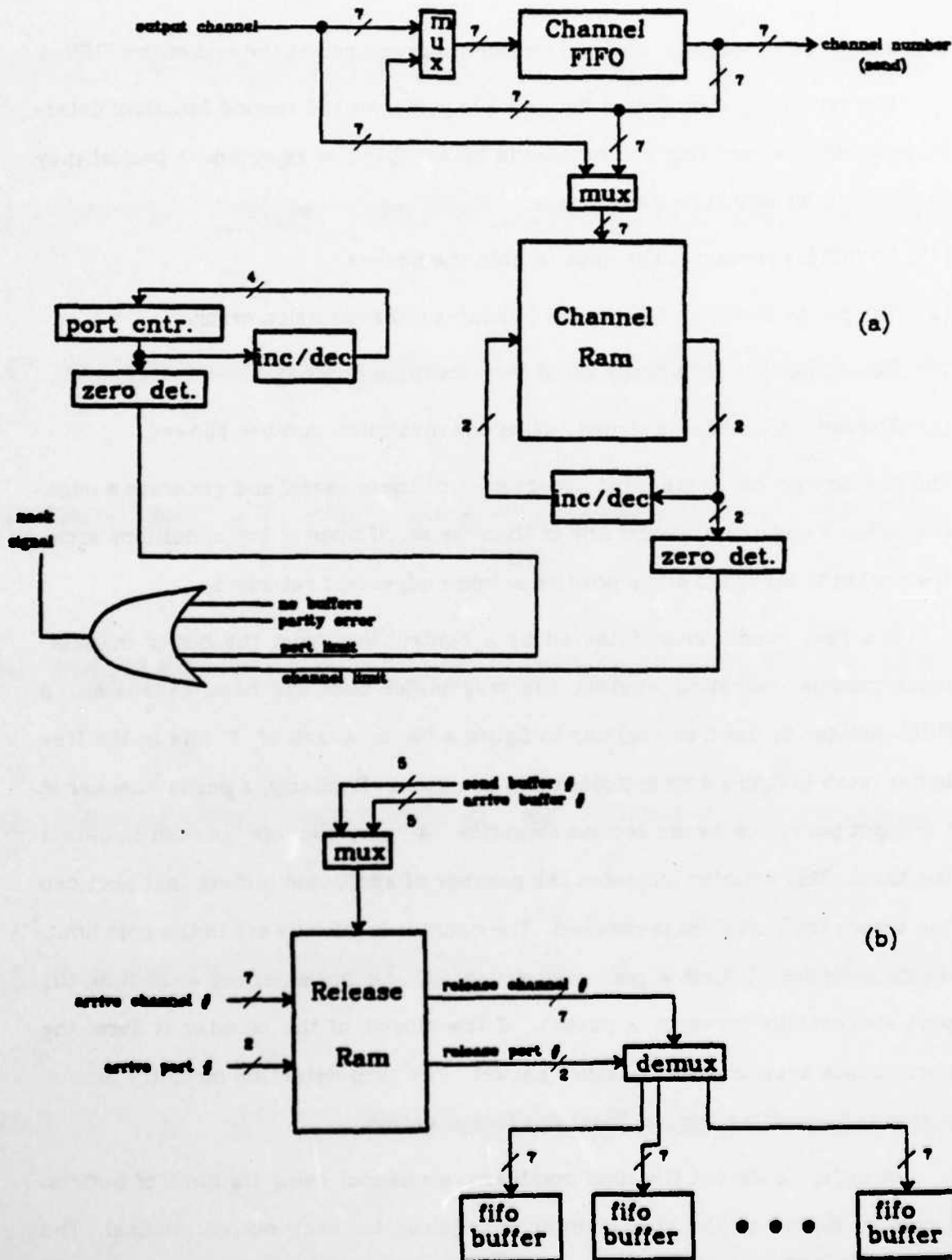
In order to prevent buffer hogging, each output channel may not use more than some "channel limit" of buffers at any one time. In addition, each output port may not use more than some "port limit" of buffers. Note that the port and channel limits only restrict the number of buffers the port and channel can use, and do not represent an a priori allocation of buffer space.

A block diagram of the flow control circuitry for one port is shown in figure 4.10a. The circuitry performs two functions:

- (1) It selects a channel which is waiting to use the link and initiates a request (to the buffer manager) to forward the next packet on this channel.
- (2) It accepts or rejects arriving packets.

Note that the flow control circuitry does not deal with buffer numbers. The buffer manager keeps track of which buffers are assigned to which channels.

The first function is accomplished by the "channel FIFO" shown in figure 4.10a: This memory lists channels with packets waiting to be forwarded. When the link is ready to forward a packet, the first element of the channel FIFO is removed. The resulting channel number is sent to the buffer management circuitry indicating that the next packet on this output channel is to be sent over the link. If this packet is accepted by the neighboring node, the FIFO element is



**Figure 4.10.** Flow control circuitry (a) send/acknowledge.  
(b) additional circuitry for remote buffer management.

discarded. Otherwise, the channel number is reentered at the end of the FIFO.

The remaining circuitry in figure 4.10a performs the second function: determine whether an arriving packet should be accepted or rejected. A packet may be rejected for any of four reasons:

- (1) No buffers remain in the node to hold the packet.
- (2) The parity check on the header indicates a transmission error.
- (3) The output port is already using the maximum number allowed.
- (4) The output channel is already using the maximum number allowed.

The flow control hardware must detect each of these cases, and generate a negative acknowledgement should any of them arise. If none of the conditions arise, the packet is accepted and a positive acknowledgement returned.

The first condition is detected by a control line from the buffer management module indicating whether the free buffer pool has been exhausted. A NULL pointer in the free register in figure 4.8a, or a lack of '1' bits in the free buffer latch in figure 4.8b indicates this condition. Similarly, a parity checker in the input port detects the second condition. A "port counter" is used to detect the third. This counter indicates the number of additional buffers that port can use before the port limit is reached. The counter is initially set to the port limit, decremented each time a packet is accepted, and incremented each time the port successfully forwards a packet. If the output of the counter is zero, the port cannot accommodate another packet. The zero-detection circuitry, implemented by a single nor gate, identifies this situation.

In order to detect the final condition, a channel using its limit of buffers, circuitry similar to the port counter is required for each output channel. The c-word "channel RAM" indicates the number of buffers each channel can use before reaching its channel limit. Entry *i* is initially set to the channel limit for



channel  $i$ , is decremented each time a packet is accepted by that output channel, and is incremented when the channel successfully forwards a packet. If a packet arrives and the corresponding channel RAM entry is zero, the packet must be rejected and a negative acknowledgement returned.

If the restriction is made that each channel can use at most one buffer at a time, then the circuitry in figure 4.10a can be simplified. The channel RAM is now one bit wide, and indicates whether or not the channel has a packet waiting to be forwarded. The increment/decrement circuit attached to the channel RAM is no longer needed, since accepting a packet implies setting a bit in the RAM, and forwarding a packet implies resetting a bit. Similarly, the channel RAM's zero-detection circuitry is not required, since only a single bit is output from the RAM.

#### 4.4.4.3. Flow Control Hardware: Remote Buffer Management

In this scheme, each output port maintains enough information about buffer allocation in its neighboring nodes to determine which channels may send packets, and which must wait. A channel must wait if it is using "too many" of its neighbor's buffers, or if there is insufficient free buffer space to hold a new packet. Control decisions are made by the sender, and receivers must accept all packets sent over the link. Eventually the receiver will forward the packet to a third node. When this happens, the receiver reports back to the sender by sending a "release channel number" indicating that the buffer is again available for another packet. This indicates the number of the channel which originally sent the packet.

In addition to restricting the number of buffers each channel can use at one time, a "port limit" restricts the number of buffers each output port can use. Both the port and channel limits are initialized by the local routing controller. If the port limit is  $pl$ , then  $pl$  buffers are reserved in the neighboring node for

use by this output port. This is in contrast to the send/acknowledge design in which the port limit only restricts the number the port can use, but does not actually reserve buffer space. Because no retransmissions are used, the remote buffer management scheme must reserve buffers to avoid overflows, since this will result in lost packets.

The flow control circuitry must perform four functions:

- (1) It selects a channel which is waiting to use the link, and initiates a request (to the buffer manager) to forward the next packet on this channel.
- (2) It generates release channel numbers to previous nodes.
- (3) It processes incoming release channel numbers.
- (4) It maintains information of buffer allocation in receiving nodes.

The physical circuitry for performing the first function is virtually the same as that shown in figure 4.10a for the send/acknowledge scheme, however the logical meaning of the information kept in the channel RAM is different. Instead of indicating the number of local buffers below the channel's limit in the local node, the RAM indicates the number of *remote* buffers below the limit in the neighboring node. As long as entry *i* is not zero, channel *i* may send another packet, assuming the port limit has not been reached. Similarly, the port counter also refers to buffers in the neighboring node used by this output port.

When a packet arrives, the number of the output channel responsible for forwarding the packet is added to the end of the channel FIFO. Since all packets are accepted, no further processing is required. To forward a packet, the next entry in the channel FIFO is removed. The corresponding channel number is used to address the channel RAM. If the corresponding entry of the channel RAM is not zero, and if the port counter is not zero, then the channel number is sent to the buffer module and the next packet on this channel is sent over the link.

The port counter and channel RAM entry are then decremented. If either of the counters was zero, the channel cannot forward the packet so the channel number is reentered at the end of the channel FIFO.

Each time a packet is forwarded, a release channel number must be sent back to the neighboring node which sent the packet. The circuitry in figure 4.10b performs this function. In order to generate release channel numbers, information must be kept with each packet that indicates which input port and channel it arrived on. A b-word "release RAM" accomplishes this task. Element  $i$  indicates the input port and channel number the packet in buffer  $i$  arrived on. This information is loaded into the release RAM when a packet arrives and read when it is forwarded. A small fifo buffer in each output port holds the channel number portion of the word until it can be forwarded to the neighbor which sent the packet.

Finally, when a release channel number is received from a neighboring node, indicating that a certain channel is using one fewer buffer, the count of buffers the channel is allowed to use must be incremented. The appropriate entry of the channel RAM is read, incremented, and written back into the RAM, completing the processing of the release.

The simplifications resulting from constraining each output channel to using at most one remote buffer at a time are similar to those described in the send/acknowledge scheme. The channel RAM is again one bit wide. Forwarding a packet resets a bit in the RAM, effectively disabling the channel. Receiving a release channel number causes the bit to be set, reenabling the channel.

#### 4.5. Evaluation of Communication Component Parameters

In order to evaluate the amount of circuitry required to implement the communication component described above, estimates are required of:

- (1) the number of I/O Ports
- (2) the number of virtual channels
- (3) the number of buffers.

Chapters 2 and 3 examined the first question in detail and concluded that from 3 to 5 I/O ports should be used. The remaining two questions will be discussed next.

#### 4.5.1. Number of Virtual Channels

Because each virtual channel requires a certain amount of overhead circuitry, the number of channels must be limited. In addition, it is desirable to limit the number of channels on each link to prevent "overbooking" the link's bandwidth, since this will lead to long queueing delays on the link and to poor performance. On the other hand, providing too few channels per link will lead to a high failure rate in establishing virtual circuits, deadlock situations, and underutilization of the link's bandwidth. Thus the number of virtual channels per link must be chosen to achieve good link utilization without incurring an excessive amount of overhead circuitry.

First, link utilization will be used to determine the proper number of channels per link. The overhead issue will be ignored for now. Deadlocks can be broken with an end-to-end timeout mechanism, as will be discussed later.

Each virtual circuit using a link requires a certain amount of bandwidth. Since the bandwidth provided by each link is fixed, each link can support a large number of circuits with low bandwidth requirements, or a small number of circuits with high bandwidth requirements. If a large number of channels are provided to accommodate the former case, a number of high bandwidth circuits may use the link and overbook the available bandwidth. If a small number of channels are provided, much of the link's bandwidth will be wasted when many

low bandwidth circuits monopolize the available channels.

One approach to resolving this dilemma is to provide enough channels to accommodate a large number of low bandwidth circuits, but to also provide a separate mechanism which prevents overbooking the link's bandwidth. New circuits may not be established on the link if the link's bandwidth has been fully booked, regardless of the number of unallocated channels remaining. Later, when existing circuits using the link are torn down, new circuits could again be established.

In order to implement this mechanism, the bandwidth requirements of each circuit must be estimated. This could be accomplished statically when the circuit is established (e.g. the operating system may be able to provide this information based on the type of traffic expected over the circuit), or dynamically, "on the fly", by measuring traffic on the circuit. Of course, the latter scheme has the disadvantage that link bandwidth may still be overbooked since the amount of bandwidth required by the circuit is not known until after it is established, i.e. a high bandwidth circuit may be established over the link before it is known that its bandwidth requirements overbook the link.

Both the static and the dynamic schemes could be implemented by associating a "link bandwidth indicator" with each output link which indicates the anticipated bandwidth requirements of circuits using the link. When this bandwidth indicator exceeds some threshold, no more traffic is routed over that link. In the first scheme, using "hints" from the operating system, a field in the packet which sets up the virtual circuit could indicate the anticipated bandwidth requirements of that circuit. The link indicator is increased by the value of this field when the circuit is set up, and decreased when the circuit is torn down. The value of this field must be included in the packet tearing down the circuit as well as the header, since the component does not keep track of the bandwidth

requirements of each channel.

In the dynamic scheme, the bandwidth indicator could be incremented each time a packet is sent on that link. Periodically, the routing controller examines the indicator to determine if the link is overbooked, and then clears it. Finally, a third alternative is to measure the average queue length on each link, and declare the link overbooked if this average exceeds a certain threshold. Again however, these two schemes do not prevent overbooking the link's bandwidth, but rather attempt to prevent a bad situation from becoming worse.

If a separate mechanism is used to prevent overloading the link, each component should ideally provide an unlimited number of channels since this guarantees that it will never needlessly block circuits trying to use a link with excess capacity. This number can be reduced however, if the minimum bandwidth requirements of any virtual circuit can be established. The maximum number of channels the link will ever require can be calculated by dividing the total link bandwidth by this minimum channel bandwidth, as will be derived below.

In this context, two questions must be considered:

- (1) How many minimum bandwidth circuits can be maintained on a fixed bandwidth link?
- (2) How much traffic corresponds to a minimum traffic load?

The first question lends itself to a precise mathematical analysis, and will be discussed next. The second is more difficult to resolve since it is application program dependent. It will be addressed later.

Given an expected traffic load on each circuit, the proper number of channels can be estimated via a queueing model. The model for the traffic load on each communication link is shown in figure 4.11. The  $n$  virtual channels using

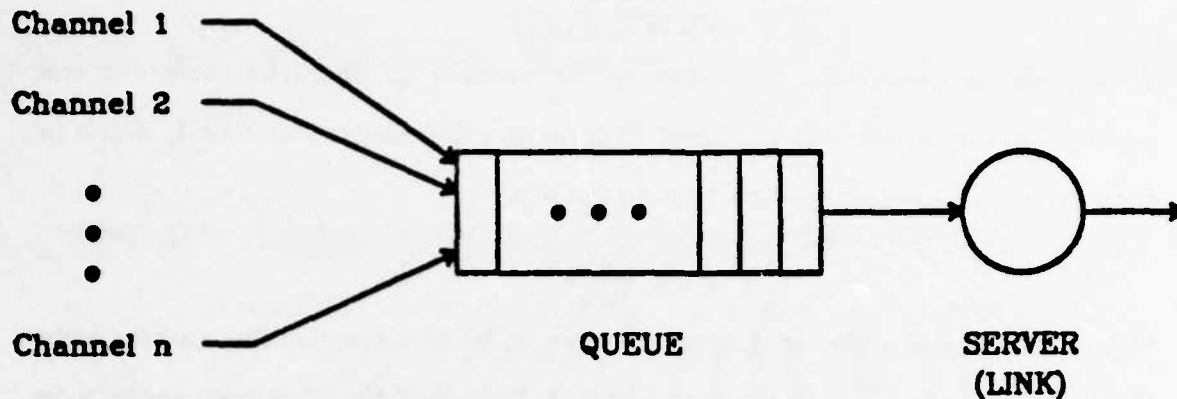


Figure 4.11. Queueing model for analyzing number of channels.

the link are modeled as a single server queue with traffic arriving from  $n$  sources. It will be assumed that packet arrival times on each virtual circuit follow a Poisson distribution. Since fixed-length packets are used, service times are deterministic, resulting in an M/G/1 queueing model. From the Pollaczek-Khinchin mean value formula [Klei75], the average time  $\bar{W}$  each packet spends in the queue waiting for the link is

$$\bar{W} = \frac{\bar{x}\rho}{2(1-\rho)}$$

where  $\bar{x}$  is the service time, i.e. the time required to transmit a packet, and  $\rho$  is the link utilization. Assuming the average arrival rate on each of the  $n$  virtual circuits is  $\lambda$  messages per second, or  $\lambda m_t$  bits per second, where  $m_t$  is the packet length in bits, the utilization of a link with a bandwidth of  $b$  bits per second is

$$\rho = \frac{n\lambda m_t}{b}.$$

Since the service time is  $\bar{x} = m_t/b$ , we find



$$N = \frac{n \lambda \pi_1^2}{2 b (b - n \lambda \pi_1)}$$

To determine numerical estimates of the number of channels, consider the number of virtual circuits required to drive the link utilization  $\rho$  to 1, which in turn drives the average waiting time to infinity:

$$n = \frac{b}{\lambda \pi_1}$$

Figure 4.12 shows a plot of this quantity for an 80 Mbit/second link (a byte wide link running at 10 Mhz) as a function of  $1/\lambda$ , the mean time between packets on each circuit. This plot also assumes that packets are 17 bytes in length.

The critical parameter in evaluating the number of channels is the expected traffic load on each virtual circuit. Unfortunately, the bandwidth requirements of each circuit may be arbitrarily small, implying an arbitrarily large number of channels should be supported. In reality however, seldomly

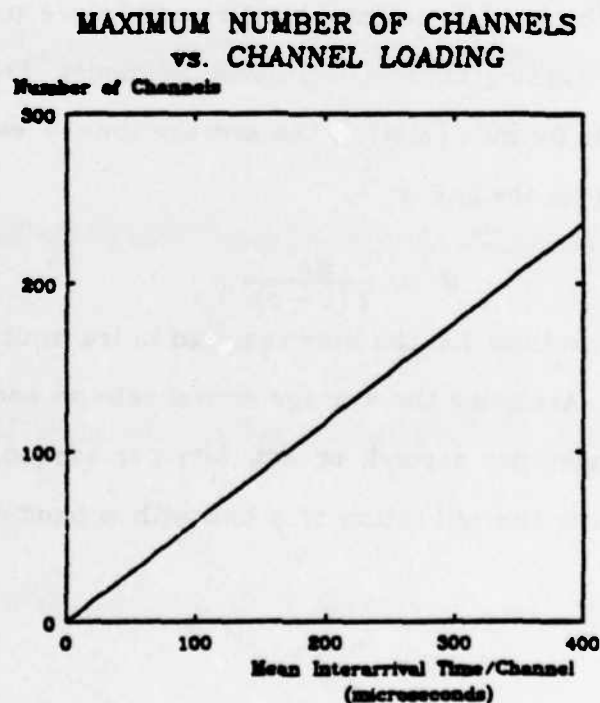


Figure 4.12. Number of channels vs. load per channel.

used virtual circuits may be torn down and reestablished as necessary to reduce the number of channels required. Since reestablishing a circuit incurs considerably more delay than sending a message on an existing circuit, these lightly loaded circuits must not have low latency requirements.

In order to determine numerical estimates of virtual circuit loading, the average time between messages on virtual circuits was measured for the five application programs described in chapter 3 (the artificial traffic load program is excluded) assuming negligible communication delays. These arrival rates are shown in table 4.1 below, and represent rates averaged over all virtual circuits in the application program weighted according to the number of messages sent on each circuit. If  $n_i$  and  $\lambda_i$  are respectively the number of messages sent and the average arrival rate on virtual circuit  $i$ , then the overall average arrival rate is computed as:

$$\lambda = \frac{1}{\sum_i n_i} \sum_i n_i \lambda_i .$$

Standard deviations for the interarrival times are also shown in table 4.1. The zero value in the FFT program is due to the regularity of its structure: All tasks iteratively perform the same computation, so the time between messages is always the same. The other values reflect the fact that the programs typically use two types of circuits - those with little or no computation between messages, e.g. the circuits distributing the initial data samples in the signal processing

Table 4.1  
AVERAGE ARRIVAL RATES PER VIRTUAL CIRCUIT

PROGRAM	ARRIVAL RATE (msgs./sec.)	INTERARRIVAL TIME (microseconds)	STANDARD DEVIATION (microseconds)	NUMBER OF CHANNELS (80 Mbit link)
BARNWELL	9940	100.6	40.4	59
FFT	17200	58.0	0.0	34
BLOCK	29200	34.2	43.8	20
JORDAN	45200	22.1	23.2	13
LU	85500	11.7	9.2	7

programs, and those with more significant computations between messages. These latter circuits have an average interarrival time which is much larger than the former.

Figure 4.13 shows the average waiting time to use an 80 Mbit/second link loaded with virtual circuits which each carry an artificial traffic load corresponding to the average values listed in table 4.1. The curves show that up to 59 channels should be allowed for the program exhibiting the lightest traffic load — the Barnwell signal processing program. This value is also listed in table 4.1 along with the corresponding values for the other application programs. Thus, for workloads similar to those discussed here, it would be reasonable to provide up to 64 channels on each link. If one considers the standard deviation on the Barnwell program, one could argue that this figure should be raised to 128, since it is possible that most of the circuits using a link could by chance fall below the average arrival rate.

As discussed earlier, the tasks in the application programs listed in table 4.1 communicate relatively frequently. Programs requiring less frequent communication use circuits that are more lightly loaded, implying each link could support even more channels. Thus, the figure derived above should be considered a lower bound rather than an absolute estimate of the number of channels. Since there may be any number of circuits which communicate infrequently, but which require low latency (making it unreasonable to tear down and reestablish the circuit each time a message is sent), more than 64 channels should be provided. However, increasing the number of channels increases the size of the channel number that must precede each packet, reducing the amount of bandwidth available for transmitting data. A compromise between these conflicting considerations is to provide 128 or 256 channels per link, since this allows more than 64 channels, but still confines the channel number over-

# WAITING TIME vs. NUMBER OF CHANNELS

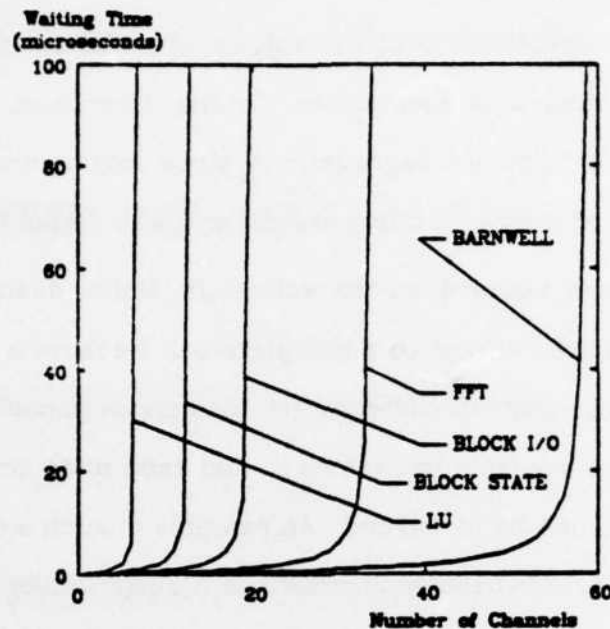


Figure 4.13. *Waiting time vs. number of channels for various programs.*

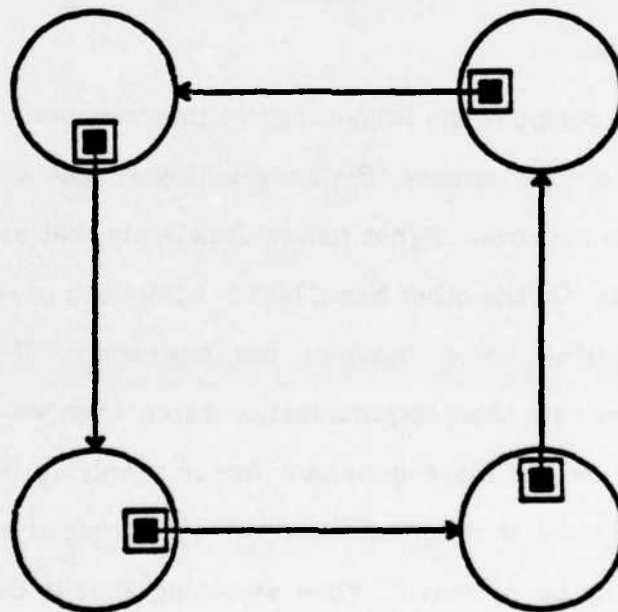
head to a single byte.

Finally, let us consider the impact future improvements in technology will have on the number of channels. Both computing speeds and link bandwidths can be expected to improve. Higher bandwidths imply that each link could support more channels. On the other hand, higher computation rates lead to higher traffic loads, implying fewer channels are necessary. If switching speeds increase at a faster rate than communication rates, then we can expect virtual circuit loading to be the more dominant factor, implying fewer channels per link. On the other hand, if communication rates progress at a higher pace, then more channels may be provided. While switching speeds can be expected to improve by an order of magnitude over the next 20 years [Keye79], fiber optic links may lead to much larger improvements, implying more channels may be supported.

#### 4.5.2. Amount of Buffer Space

Technological capabilities limit the amount of buffer space that can be provided by each communication component. On the other hand, insufficient buffer space will lead to performance degradations, since communication bandwidth is wasted and delays increased if buffers are not available to hold arriving packets.

In extreme cases, buffer deadlock will result. Buffer deadlock is a situation in which message traffic comes to a complete halt because a set of nodes have exhausted all of their available buffer space. Each node cannot forward a packet because no buffer is available to receive it, and each node cannot free a buffer because no packets can be forwarded. An example of such a deadlock situation is shown in figure 4.14, where each node has a single buffer holding a packet waiting to be forwarded. The network will remain deadlocked until a packet is discarded, releasing a buffer.



**Figure 4.14.** *Example of buffer deadlock.*

Thus, sufficient buffer space must be provided to:

- (1) avoid buffer deadlock.
- (2) ensure good performance.

Each of these issues will now be discussed in turn, followed by results from simulation studies that help to determine the buffering requirements of each communication component.

#### 4.5.2.1. Buffer Space: Deadlock Considerations

Buffer deadlock can be prevented if enough buffer space is provided in each communication component. A brute force solution is to provide each virtual channel with its own buffer. Since each circuit is allocated a buffer in each node it passes through, traffic on a circuit cannot be blocked by traffic on other circuits, so buffer deadlock cannot occur. Providing a separate buffer on each channel is wasteful however, since each component will have to provide as many buffers as there are channels. It will be seen that virtually the same performance can be achieved if many channels share a much smaller pool of buffers.

An alternative approach to avoiding buffer deadlock is outlined in [Mer180a, Mer180b]. Here, each node must have at least  $\bar{H}_{\max}$  buffers, where  $\bar{H}_{\max}$  is the maximum number of hops traversed by any virtual circuit. The buffer pool in each node is partitioned into  $\bar{H}_{\max}$  disjoint pools or levels, say  $1, 2, \dots, \bar{H}_{\max}$ . Each node maintains a "hop count" for each circuit passing through the node indicating the number of hops the circuit has traversed from the source node to the current node. The hop count is set when the virtual circuit is first established. A packet arriving at a node on a circuit with hop count  $i$  may only be placed in one of the buffers in level  $i$ . It can be shown that buffer deadlock will never occur in this scheme.

The central disadvantage of this scheme is that large networks require more buffers than smaller ones, so the communication component must provide enough buffers to accommodate the largest network it will ever become a part of. This may require an excessively large amount of buffer space. In addition, if traffic is highly localized, many buffers are wasted because those reserved for higher hop counts are never utilized.

Partitioning the buffer pool also adds a certain amount of complexity to the component. The partitioning can be accomplished by limiting the total number of buffers used by circuits at the same level, rather than physically partitioning the buffer space. For example, if  $\bar{H}_{\max}$  is 10 and there are 20 buffers in the buffer pool, it is sufficient to use the buffer management schemes described above, and ensure that the circuits at any given level collectively use no more than 2 buffers at one time. This could be implemented with a counter at each level indicating the number of free buffers currently available at that level. A packet is rejected if no more buffers are available at its particular level.

Implementation of this scheme with a remote buffer management policy for flow control is more difficult, since different nodes compete for the buffers in each level. The buffers in each level must be further partitioned among the neighboring nodes to avoid overflow within each level.

A third approach to resolving the deadlock issue is to allow deadlock to occur, but to incorporate a mechanism that ensures that deadlocks are broken. Since detecting and breaking a deadlock may be a time consuming operation, enough buffer space should be provided to ensure that deadlocks occur infrequently. The deadlock breaking mechanism could be implemented as a side effect of an end-to-end protocol using timeout counters to retransmit lost messages. In such a scheme, each message sent over a virtual circuit must be acknowledged by the receiver. If an acknowledgement is not returned after a cer-

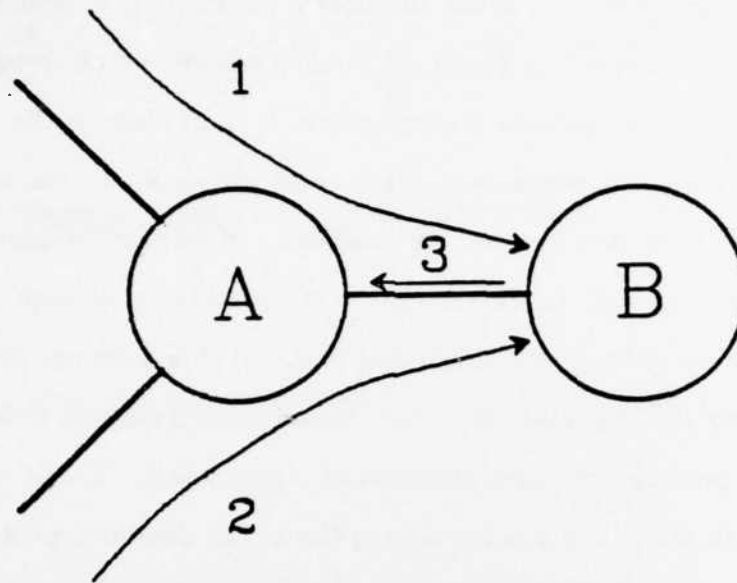


tain amount of time, the sender assumes that the message was lost and must be retransmitted. In order to avoid duplicate packets, the sender must first "clear" the virtual circuit by sending a special packet which flows through the circuit and destroys all packets it encounters. It then resends the lost message. If deadlock occurs, timeouts will result and circuits will be cleared. This releases buffer space and breaks the deadlock. Since such a timeout mechanism is already required to retransmit lost packets, the deadlock breaking mechanism incurs virtually no additional cost. In this scheme, communication components are not required to ensure that buffer deadlock never occurs, so they need not provide excessive amounts of buffer space. Thus this mechanism appears to be an attractive one for solving the buffer deadlock problem.

It might be noted that a similar mechanism could be used to break deadlocks arising when links use all of their virtual channels. Expiration of an end-to-end timeout during the set-up of a virtual circuit could trigger the release of a special packet which destroys the partially completed circuit, releasing channels, and breaking the deadlock. This protocol also requires an end-to-end acknowledgement to mark the establishment of each virtual circuit.

*Simulation experiments* similar to those discussed in chapter 3 were carried out to evaluate the number of buffers each communication component should provide to avoid buffer deadlock. The six application programs discussed in chapter 3 were executed on Simon with a switch model for a hexagonal lattice network built from 4-port communication components (figure 3.14a).

The first set of experiments assumed that each component provides  $b$  buffers, and that no restrictions are made on buffer sharing, i.e. virtual circuits may use as many buffers as are available. A large number of buffers, over 100 for some of the programs, was required in each component to avoid buffer deadlock.



**Figure 4.15.** *Example of congestion leading to deadlock.*

*Buffer hogging* was at the root of these deadlock problems. Consider the situation in figure 4.15. Virtual circuits 1 and 2 join at node A, sharing the link from A to B, and virtual circuit 3 uses the link from B to A. Suppose all three circuits carry a steady stream of packets, or equivalently, suppose a burst of packets simultaneously arrives on each circuit. Since the flow of packets into node A on circuits 1 and 2 exceeds the flow from A to B (the latter is limited by the capacity of the link from A to B) a queue begins to form at node A. The queue will grow until the free buffer pool in node A is exhausted. When this happens traffic on circuit 3 is blocked, and a queue of packets begins to grow in node B. Eventually, B's free buffer pool will also be exhausted, blocking traffic on circuits 1 and 2. The network is now deadlocked, and will remain in this state until packets are discarded.

The scenario described above can be avoided if precautions are taken to avoid buffer hogging, e.g. by restricting the number of buffers each circuit can

use. The simulation experiments were repeated assuming each circuit could not use more than one buffer in each node at one time. It was found that 12 buffers were sufficient to avoid buffer deadlock in all six application programs.

The simulation experiments thus demonstrate the need to provide a flow control mechanism which prevents buffer hogging. The studies also indicate that, it is reasonable to provide each component with a few tens of buffers, say 32, to reduce the probability of buffer deadlock.

#### 4.5.2.2. Buffer Space: Performance Considerations

Each communication component must provide enough buffer space to maintain a steady flow of traffic through the node. Otherwise, communication bandwidth will be wasted: In the send/acknowledge flow control scheme, retransmissions are required to resend rejected packets, while in the remote buffer management scheme, links simply become idle. How many buffers are required to maintain this flow? Studies of multistage permutation networks, called delta networks, indicate that virtually no performance improvement arises beyond three buffers per node [Dias81a, Dias81b]. Three buffers are not sufficient however, to avoid many deadlock situations. Thus, based on these studies, deadlock rather than performance optimization should be used to determine the amount of buffer space required.

Simulation experiments were carried out to evaluate the following questions:

- (1) How many buffers should each component provide to achieve good performance?
- (2) How many buffers should each virtual circuit be allowed to use at one time?
- (3) How well does the simple send/acknowledge flow control mechanism perform?

Before discussing the results of these simulation experiments, let us anticipate the answers to these questions by deriving an intuitive understanding of the impact of buffering and flow control on network performance.

Buffering and flow control questions are of little consequence when the network is lightly loaded, since buffering requirements are low (buffers start to empty while they are being filled) and throttling mechanisms are not necessary. Therefore, we will only consider the case in which links are congested. An example of a congested link is shown in figure 4.16a. Two circuits, 1 and 2, arrive at a node on links A and B respectively, and share link C. Assume that each circuit carries a continuous stream of packets. Since the combined bandwidths of links A and B is twice that of C, the latter becomes the bottleneck which limits the performance (i.e. bandwidth) of each circuit.

First, let us consider the simplest communication component design in which a send/acknowledge protocol is used for flow control, and where each channel is allowed to use only a single buffer at one time. Figure 4.16b indicates the utilization of the communication links over time. The contents of the buffer held by each circuit is also shown. Successive packets on virtual circuits 1 and 2 are labelled 1a, 1b, 1c ..., and 2a, 2b, 2c, ... respectively. As shown in figure 4.16b, the congested link, C, is fully utilized, and carries packets from both virtual circuits. The end-to-end bandwidth of the two virtual circuits will be equal to half the bandwidth of the C link, assuming traffic in other nodes does not limit performance. Note that most of the packets sent over links A and B must be retransmitted, since the first attempt fails because of the "one buffer per channel" restriction. Yet, these retransmissions do not waste bandwidth on the bottleneck link, C, so the end-to-end bandwidth is not affected. However, the negatively acknowledged packets do reduce the effective bandwidth of other circuits which do not use link C, but which are instead limited by the bandwidth of

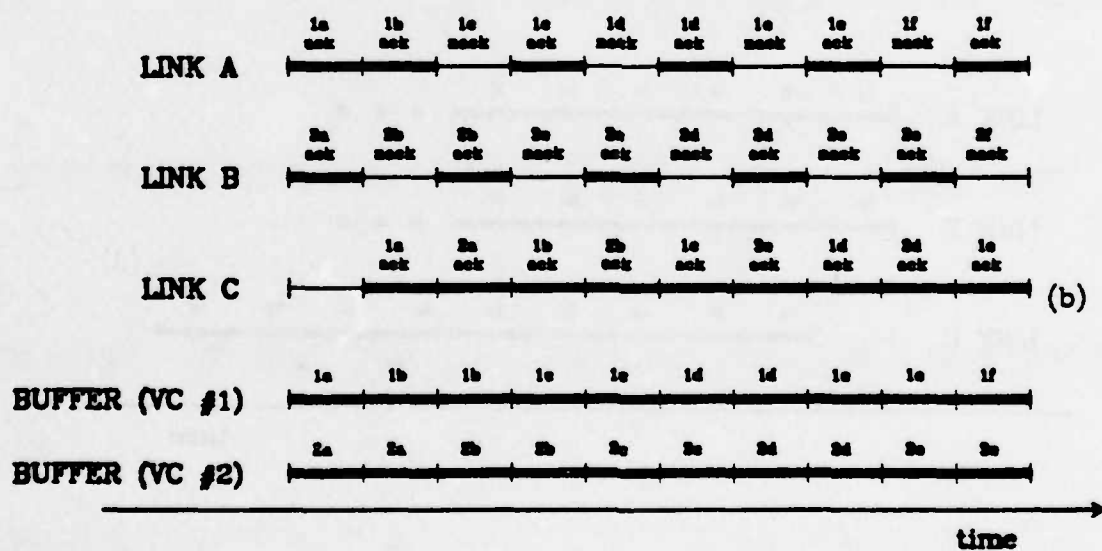
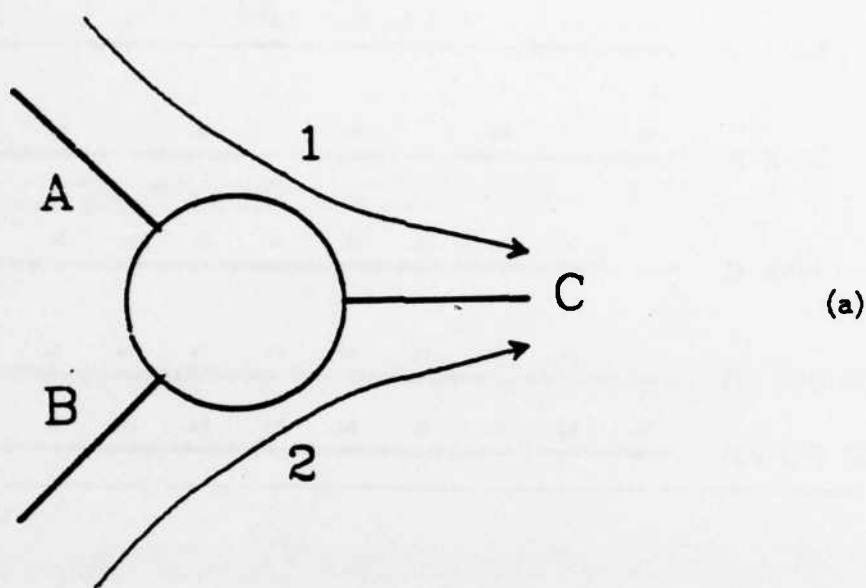


Figure 4.16. Link usage scenarios (a) Congested link.  
(b) Send/acknowledge protocol, 1 buffer/channel.

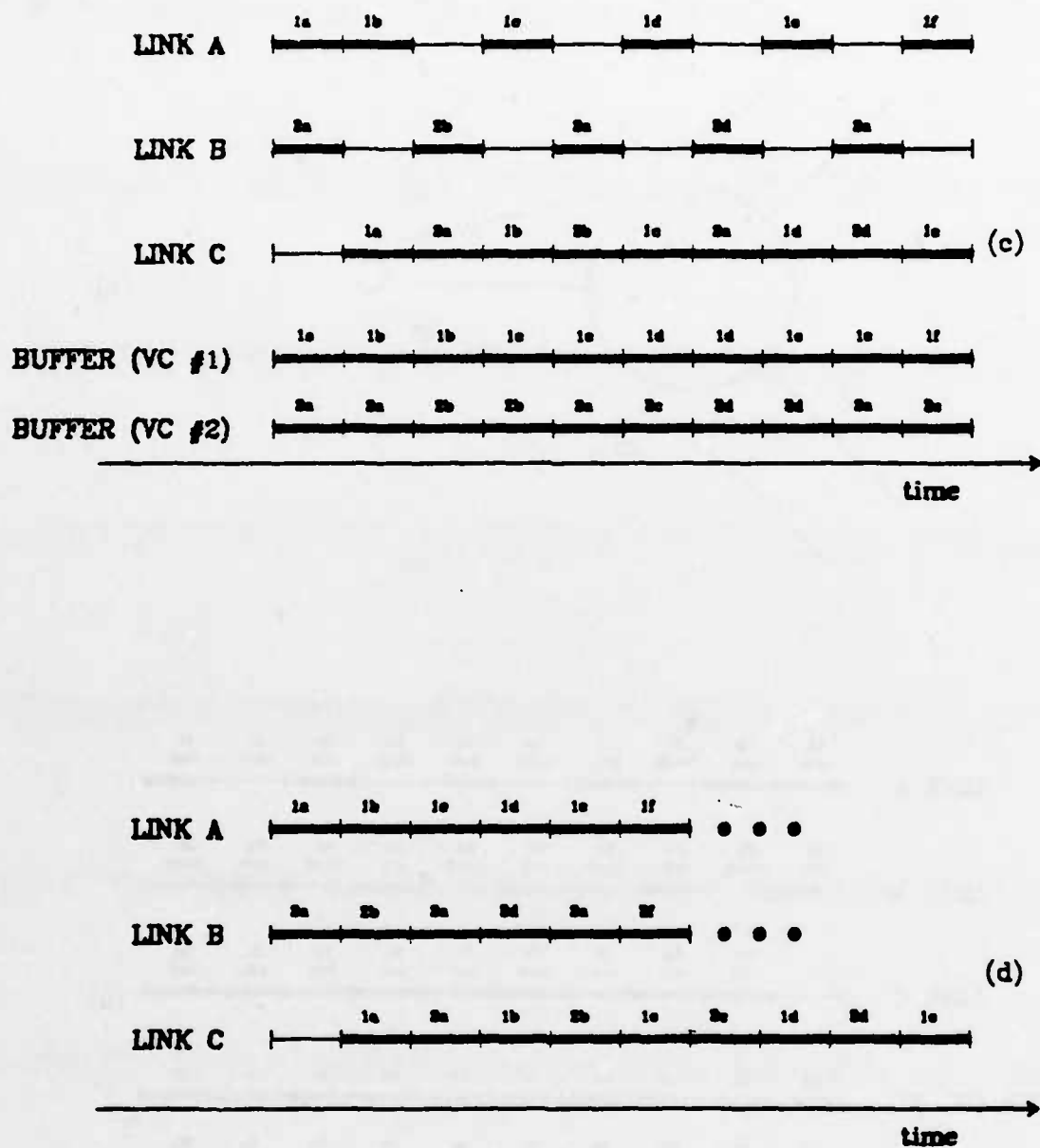


Figure 4.16. Scenarios (c) Remote buffer management, 1 buffer/channel. (d) Unlimited buffer space.

links A or B. This will be discussed later.

In figure 4.16c, the send/acknowledge protocol is replaced by a remote buffer management scheme, and circuits are still restricted to using a single buffer at one time. The optimistic assumption is made that each node has "perfect information" about buffer utilization in its neighboring nodes, i.e. the time required to transmit the feedback signal indicating a packet has been forwarded (and consequently, a buffer has been release) is assumed to be negligible. The flow of traffic over the bottleneck link, C, is exactly the same as that when the simpler send/acknowledge protocol is used, so the end-to-end bandwidth of the two virtual circuits remains the same. In comparing figures 4.16b and 4.16c, it is seen that the negatively acknowledged packets on links A and B in figure 4.16b are replaced by idle periods in figure 4.16c.

Finally, in figure 4.16d, an unlimited amount of buffer space is provided in each node. No flow control mechanism is required since there are no buffer overflows, and therefore no reason to throttle traffic. Utilization of the bottleneck link is the same as that of the previous two cases.

Intuitively, buffering is provided in each node to achieve higher network bandwidth by maintaining a large enough "backlog" of traffic in the node so that its output links remain busy under heavy traffic loads. If a node provides enough buffers to keep its links busy, then additional buffers do not improve performance. As demonstrated in figures 4.16b-d, a relatively small amount of buffer space per node is required to perform this function, explaining the results observed in [Dias81a, Dias81b]. Protection against buffer hogging must be provided however, e.g. by limiting the number of buffers each port can use, to ensure that one link does not monopolize the buffer pool and cause other link(s) to become idle.



Figures 4.16b-d indicate that the one buffer per channel restriction does not have a significant impact on network performance. Performance is limited by the bandwidth of bottleneck links, rather than a lack of buffer space. If links are underutilized, then each channel need only provide a single buffer to maintain a steady flow of traffic, as discussed earlier. If links are congested, then there is enough traffic on other circuits to keep the links busy, so no bandwidth is wasted. Performance is not improved by allowing channels to use additional buffers.

These studies also indicate that the send/acknowledge protocol does not adversely affect performance on bottleneck links (link C in figure 4.16b). This flow control mechanism does however, require retransmissions on the links leading up to the bottleneck, implying some wasted bandwidth. Often however, it is the bottleneck link which limits performance, and not the links leading up to the bottleneck, so this wasted bandwidth is of secondary importance. In addition, this waste is only of consequence if there is other traffic waiting to use the link. If other traffic exists, then the amount of wasted bandwidth is reduced, since many of the "negative acknowledgment slots" in figure 4.16b will be replaced by traffic on other virtual circuits, assuming the blocked circuit is not allowed to dominate use of the link. Thus, a more sophisticated flow control mechanism which avoids negatively acknowledged packets, e.g. the remote buffer management scheme described earlier, leads to an even smaller improvement in performance. Since network bandwidth can be improved at a higher level by using more switching chips, the additional complexity of the remote buffer management scheme is difficult to justify. In addition, output port buffer hogging is more difficult to prevent with this more sophisticated scheme (section 4.1.3), so performance may actually be reduced if this scheme is used.

Let us now examine the simulation results to see if they are consistent with the discussion presented above. Figures 4.17a-f show the performance resulting from executing the six application programs discussed in chapter 3 on a hexagonal lattice network built from 4-port communication components. All curves use a send/acknowledge protocol for flow control. If several packets are queued, waiting to use the same link, a round-robin algorithm is used to select the next packet to be sent over the link. This prevents a blocked virtual circuit from dominating use of the link by continuously retransmitting negatively acknowledged packets.

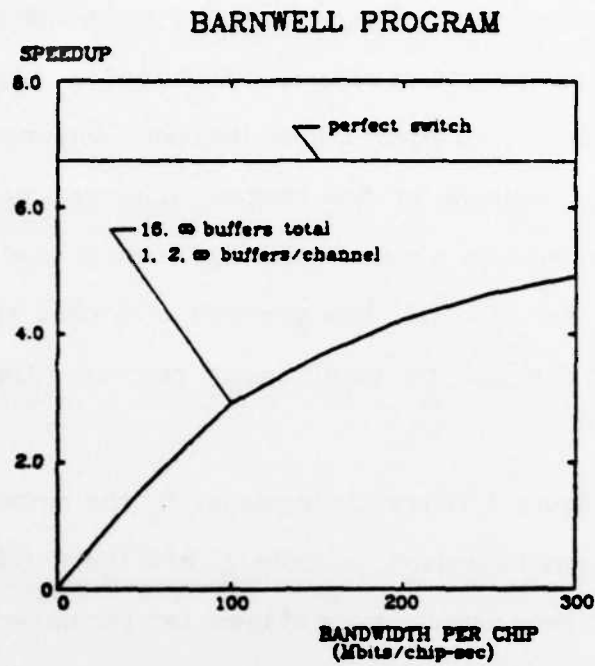
The curves in figure 4.17 are distinguished by the amount of buffer space provided in each communication component, and the degree to which buffer usage is restricted. Four combinations of these two parameters result:

- (1) Unlimited buffer space and no restrictions on buffer usage.
- (2) Unlimited buffer space but with restrictions on buffer usage.
- (3) Limited buffer space and no restrictions on buffer usage.
- (4) Limited buffer space but with restrictions on buffer usage.

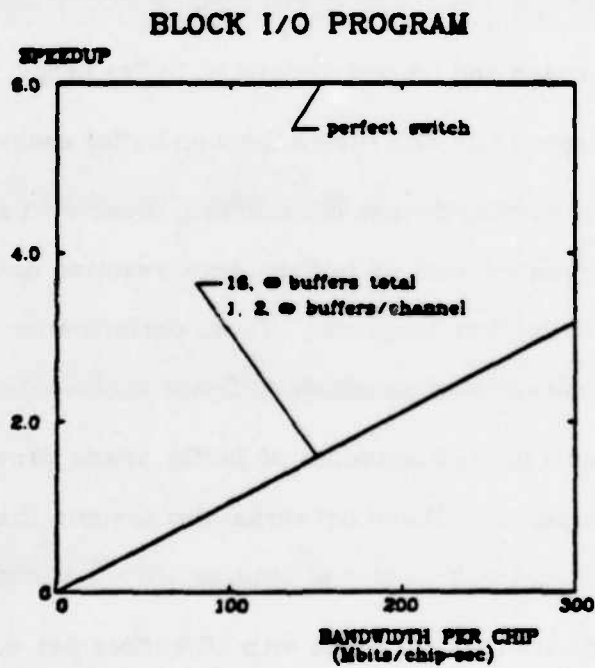
As discussed earlier, the third class of networks, those with a limited amount of buffer space and no restrictions on buffer usage, resulted in deadlock situations for many of the application programs. Thus, performance of the application programs on these networks is not shown in figure 4.17.

All networks with limited amounts of buffer space provide 16 buffers per communication component. These networks also assume that each output port may not use more than 8 buffers at one time, or  $16/\sqrt{4}$  as suggested in [Irla78].

The curves indicate that networks with 16 buffers per component yield virtually the same performance as networks with an infinite amount of buffer space, in agreement with the discussion presented earlier. Restricting virtual

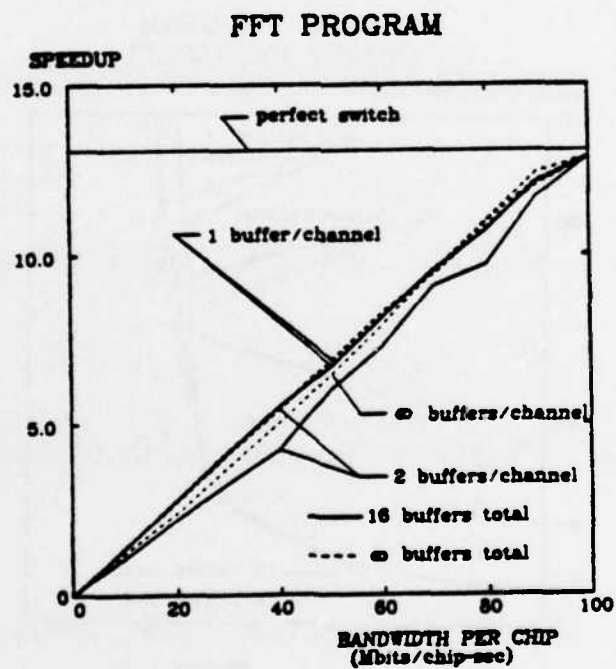
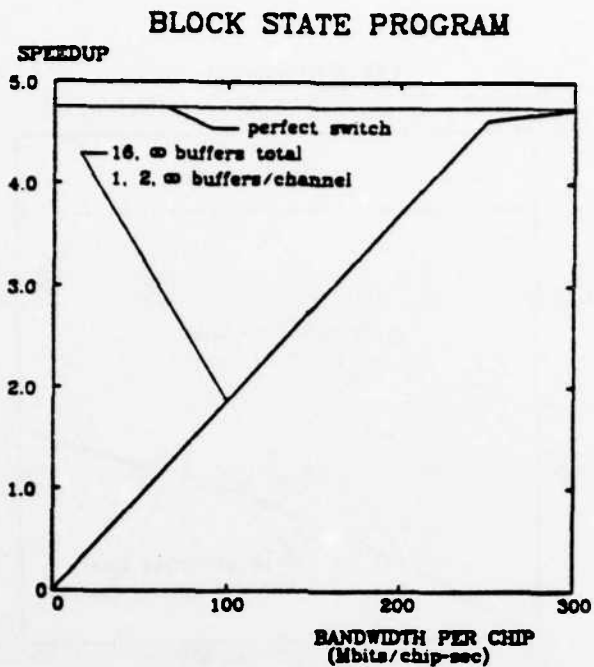


(a)



(b)

Figure 4.17. Limited buffer space (a) Barnwell. (b) Block I/O.



**Figure 4.17.** Limited buffer space (c) Block State. (d) FFT.

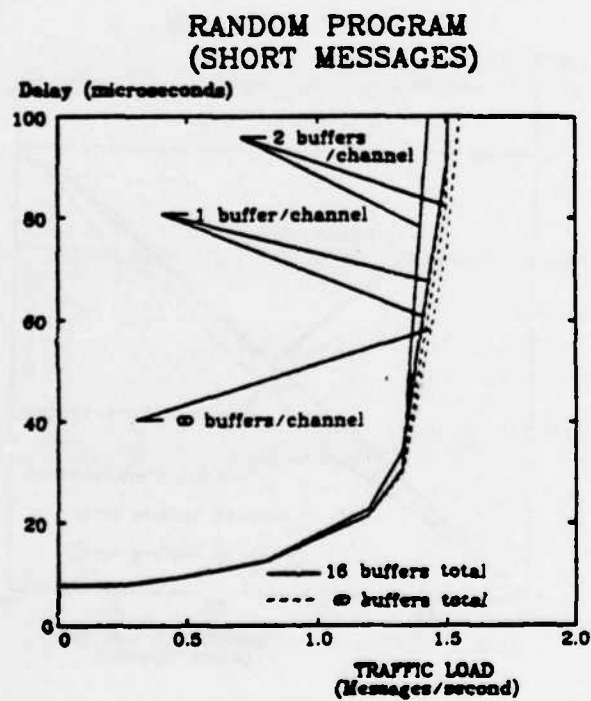
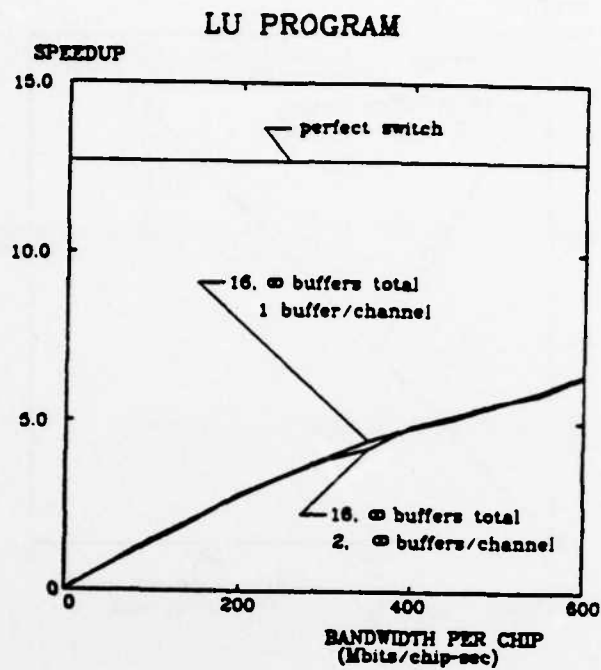
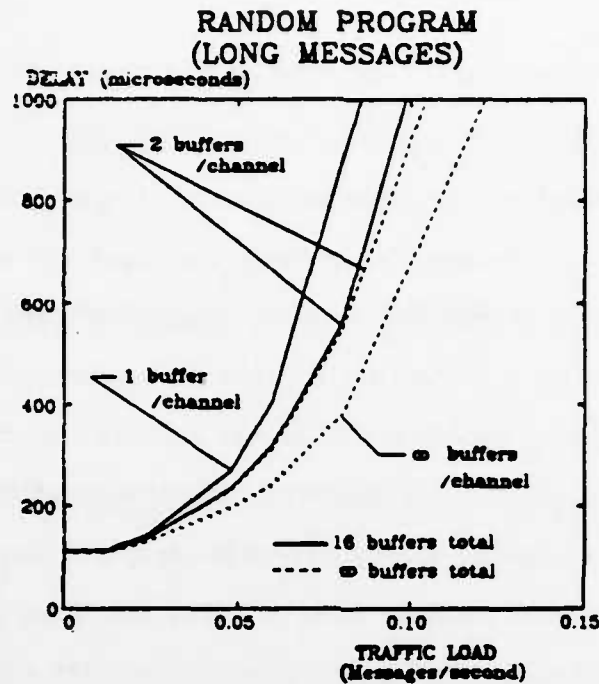


Figure 4.17. Limited buffer space (e) LU. (f) Random (short messages).



(g)

**Figure 4.17.** *Limited buffer space (g) Random (long messages).*

circuits to using at most one buffer at a time results in no significant degradation in performance. Networks using a send/acknowledge protocol for flow control yield virtually the same performance as networks with an infinite amount of buffer space and no restrictions on buffer usage. This indicates that the bandwidth wasted by negatively acknowledged packets does not have a significant effect on performance. This is due in part to the round-robin algorithm used for scheduling usage of the communication links. Blocked virtual circuits relinquish use of the link when a packet is rejected, allowing other traffic to use the link. Thus, the simulation results agree with the intuitive arguments presented earlier.

It might be noted that many of the programs achieve identical speedups regardless of the amount of buffering provided, or the buffer restrictions enforced. In these programs, a single, or a few isolated bottleneck links limit performance, e.g. the SISO programs are limited by the links carrying the initial data samples. This is in agreement with the scenarios outlined in figures 4.16b-

d, where identical utilizations are achieved on bottleneck links regardless of the buffering scheme used.

All of the application programs send small, single-packet messages. To examine buffering requirements when large messages are used, the artificial traffic load program was modified to send longer messages consisting of 256 bytes, or 16 packets each. Figure 4.17g shows the performance of this program under different buffering schemes. The curves indicate that message delays are the same under light traffic loads, demonstrating that one buffer per virtual circuit is adequate to maintain a steady stream of packets if there is no interfering traffic. The curves also indicate however, that networks achieve somewhat higher bandwidth if multiple buffers per virtual circuit are allowed. As buffering is increased, the number of negatively acknowledged packets on circuits leading up to congested links is reduced, and bandwidth improves. This additional performance must be weighed against the added complexity of allowing multiple buffers per virtual circuit. In light of the fact that network bandwidth can be improved by increasing the number of communication chips, it is doubtful that this additional improvement is justifiable. In addition, the additional complexity of allowing a circuit to use more than one buffer (a fifo queue on each channel is required) may lead to longer circuit delays, and slower clock rates, reducing performance.

#### **4.6. Complexity of the Communication Component**

Using the results presented above, the complexity of the VLSI communication components described here can be estimated. It is assumed that each component provides from 64 to 256 channels per link, 32 buffers, each large enough to accommodate 16 bytes of data, and 4 I/O ports. Transistor counts for different versions of communication components providing different levels of functionality are presented.



The communication component design described here consists of 6 modules:

- (1) I/O ports,
- (2) routing controller,
- (3) translation tables,
- (4) packet buffers,
- (5) buffer management circuitry,
- (6) flow control circuitry.

Each of these will be discussed in turn. Estimates of the amount of circuitry required for these modules are summarized in table 4.2.

Approximate gate counts are derived in part from the designs described in the TTL Databook [Inst76]. For example, the five 2 line to 1 line multiplexers shown in figure 4.10b are assumed to require 18 gates, based on an extension of the corresponding TTL part, the SN74157 [Inst76]. Similarly, registers are assumed to use 5 gates per bit. Data paths for the high speed buses and communication links are 8 bits wide. It is assumed that each finite state machine requires 200 gates. This is based on the average number of gates required for the finite state machines described in [Fuji80], which are of roughly the same complexity as those described here. Finally, the transistor counts assume four transistors are required for each gate. The estimates in table 4.2 are rounded to the nearest 1000 transistors.

Estimates for the I/O ports are based on the circuitry described in [Laur79]. The output port estimate includes circuitry for removing data from the high speed bus and driving the communication links. The input port estimate includes circuitry for driving the high speed bus, as well as three temporary registers to handle conflicts in accessing the shared memory modules, as

**Table 4.2**  
**Transistor Counts**

module	logic (transistors)	memory (kbits)		
	all designs	64 channels	128 channels	256 channels
I/O Ports (4 ports)	8000	-	-	-
Routing controller				
simple	9000	0.6	1.1	2.1
complex	10000	1.4	1.9	2.9
Routing Processor	20000	32.0	32.0	32.0
Translation Table	-	2.2	5.0	11.0
Buffers (16 modules)	8000	4.0	4.0	4.0
Buffer Management				
≤1 buffer/vc	2000	1.3	2.5	5.0
≤4 buffers/vc	2000	2.7	5.2	10.2
Flow Control				
Send / Acknowledge				
≤1 buffer/vc	11000	0.6	0.9	1.5
≤4 buffers/vc	12000	0.9	1.4	2.5
Remote Buffer				
≤1 buffer/vc	13000	1.0	1.3	2.0
≤4 buffers/vc	14000	1.2	1.8	3.0
totals				
simplest	58000	40.7	45.5	55.6
most complex	62000	43.5	49.9	63.1

discussed earlier. Based on this, each I/O port requires roughly 2000 transistors, or 8000 transistors for 4 ports.

The routing controller estimates are based on the design described in [Fuji80], modified to include circuitry for a 256 entry hierarchical routing table supporting up to 8 levels of lookup tables (see figures 4.4a and 4.4b). The numbers in table 4.2 only include the hardware support provided by the routing controller for setting the translation tables, and do not include the processor or microcode memory portions of the routing controller. These are listed separately in the table under "routing processor".

The translation table requires  $p \times c$  entries. Each entry specifies an output port or the routing controller (3 bits) and a channel number (6, 7, or 8 bits for 64, 128, or 256 channels respectively), so 9, 10, or 11 bits are required. The buffer memory estimate includes circuitry to interface each memory module to

the two buses, registers to hold the pipelined buffer addresses, and a 32 byte RAM to hold data. The figure in table 4.2 includes circuitry for sixteen memory modules.

Two estimates of the buffer management circuitry are shown in table 4.2. The first assumes each virtual circuit can only use at most one buffer at a time, and is based on the design shown in figure 4.8b. The second assumes four buffers can be used, and is based on the design in figure 4.8a.

Four estimates of the flow control logic are shown. The first two use the send/acknowledge protocol, and the latter two use a remote buffer management scheme. In addition, each of these designs allows virtual circuits to use either one, or up to four buffers at a time. The estimate for the send/acknowledge protocol with up to four buffers per channel is based on the design shown in figure 4.10a. As discussed in section 4.2.4.3, the remote buffer management scheme uses the same circuitry, as well as the additional logic shown in figure 4.10b. Modifications corresponding to the one buffer per channel restriction are outlined in sections 4.2.4.2 and 4.2.4.3.

The figures in table 4.2 indicate that a minimal complexity communication component with 4 I/O ports, 32 16-byte buffers, 64 channels per link, one buffer per virtual circuit, and a send/acknowledge flow control scheme, can be constructed with approximately 58,000 transistors for logic, and 40.7 kbits of RAM. Assuming single transistor ROM cells for microcode memory and single transistor dynamic RAM cells, approximately 100,000 transistors are required. Except for the number of channels, this design is in accordance with the design recommendations derived throughout this chapter. A similar design using 256 channels per link and the more complex routing scheme requires 59,000 transistors of logic, and 56.4 kbits of RAM. Assuming a static RAM implementation using 5 transistors per RAM cell, this more complex design requires on the order of

350,000 transistors, most of which is taken up by memory. Chips are currently available using 450,000 transistors, so the communication components described here can be implemented with current integrated circuit technology.

Figure 4.18 shows a possible floorplan for the 64 channel communication component described above. Sizes of various sections of the chip are based on the approximate transistor counts listed in table 4.2, based on single transistor RAM and ROM cells. Data paths correspond to those shown in figures 4.6 and 4.7.

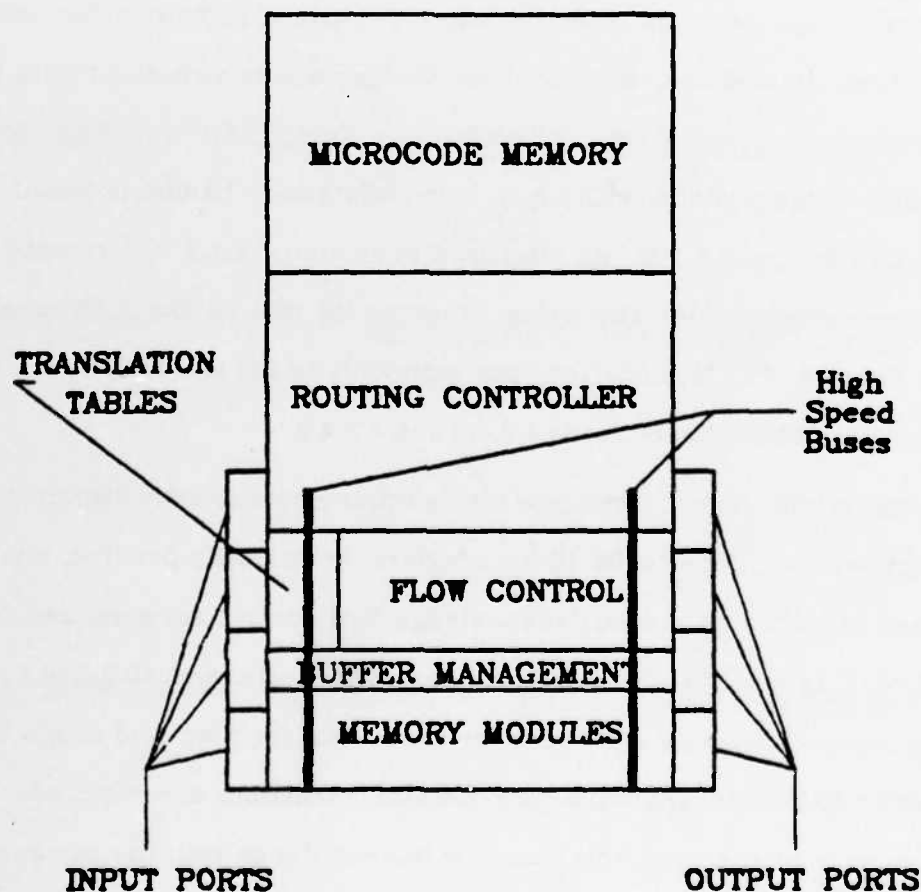


Figure 4.18. Floorplan for communication component.

## CHAPTER FIVE

### CONCLUSIONS

VLSI technology can provide us with a novel set of building blocks for the construction of high-performance point-to-point networks for closely coupled multicomputer systems. "Plug-compatible" VLSI communication components with 3 to 5 ports make particularly attractive building blocks. Their modularity permits the incremental growth of a multicomputer system with a corresponding growth of the total bandwidth of the communication domain. To be useful for the construction of systems with hundreds or thousands of processors, the complexity of these components must be above a certain threshold. The functionality of MOS VLSI chips now exceeds this threshold.

Technological considerations in the design of communication components have been examined. The described approach based on dedicated links between individual switching nodes is well matched to the evolving VLSI MOS technology. The overall performance of the network depends critically on the total chip bandwidth of these components, which is determined to a large degree by packaging technology. Performance is also influenced by the buffering and forwarding policies employed, which depend themselves on the amount of buffer space and the complexity of the control logic in the switching components. Analytic and simulation models have been used to investigate the impact of these considerations on overall network performance. Based on these studies, a number of conclusions can be drawn regarding the design of VLSI communication components. These include:

- (1) A small number of ports should be used, say from 3 to 5.

- (2) Under current technology, the degree of multiplexing on each communication link should be relatively large. Each link should provide a large number of channels, say 128 or 256.
- (3) Only a relatively small number of buffers, say 16 or 32, need to be provided. Further, restricting virtual circuits to using at most one buffer per node at one time causes little performance degradation.
- (4) Negatively acknowledged packets in the send/acknowledge flow control mechanism do not lead to a significant performance degradation, implying that more sophisticated schemes, such as the sender-controlled remote buffer management scheme, are not justified.
- (5) A multicast mechanism has a significant impact on performance in applications which send the same data to many different destination processors.

An important issue that has *not* been addressed by this thesis concerns fault tolerance. If a communication component fails, routing tables need to be updated, and broken message paths must be restored. The rerouting must be done in a manner that ensures that loops are not introduced. Much of the work in rerouting strategies in computer networks is directly applicable here [Taji77, Merl79, Sega81]. One must also safeguard the network against messages addressed to non-existent or unreachable nodes. These issues cannot be ignored in any communication component design.

For the near future, the limited number of devices that can be fabricated economically on a single chip will encourage the development of separate switching components. However, towards the end of this decade, the preferred building block may well consist of a powerful processor, a substantial amount of on-chip memory, and the switching circuitry that is needed so that these components can be readily plugged together into a working multicomputer system.

## REFERENCES

- [Acke65] S. B. Ackers, "On the Construction of (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-14 p. 488 (June 1965).
- [Adam82] G. Adams III and H. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers* C-31(5) pp. 443-454 (May 1982).
- [Ahuj82] V. Ahuja, *Design and Analysis of Computer Communication Networks*, McGraw-Hill, New York (1982).
- [Amar83] D. Amar, "On the Connectivity of some Telecommunication Networks," *IEEE Transactions on Computers* C-32(5) pp. 512-519 (May 1983).
- [Arde81] B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Transactions on Computers* C-30(4) pp. 291-295 (April 1981).
- [Arde82] B. W. Arden and H. Lee, "A Regular Network for Multicomputer Systems," *IEEE Transactions on Computers* C-31(1) pp. 60-69 (Jan. 1982).
- [Baas78] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Massachusetts (1978).
- [Barn80a] C. Barnes and S. Shinnaka, "Block Shift Invariance and Block Implementation of Discrete-Time Filters," *IEEE Transactions on Circuits and Systems* CAS-27 pp. 667-672 (August 1980).
- [Barn80b] C. Barnes and S. Shinnaka, "Finite Word Effects in Block-State Realizations of Fixed-Point Digital Filters," *IEEE Transactions on Circuits and Systems* CAS-27 pp. 345-349 (May 1980).
- [Barn78] T. P. Barnwell III, S. Gaglio, and R. M. Price, "A Multi-Microprocessor Architecture for Digital Signal Processing," *Proc. of the 1978 Intl. Conf. on Parallel Processing*, pp. 115-121 (August 1978).
- [Barn79] T. P. Barnwell III, C. J. M. Hodges, and S. Gaglio, "Efficient Implementations of One And Two Dimensional Digital Signal Processing Algorithms on a Multiprocessor Architecture," *1979 Intl. Conf. on ASSP, Washington, D. C.*, pp. 698-701 (Apr. 1979).
- [Barn82] T. P. Barnwell III and C. J. M. Hodges, "Optimal Implementation of Signal Flow Graphs on Synchronous Multiprocessors," *Proc. of the 1982 Intl. Conf. on Parallel Processing*, pp. 90-95 (August 1982).



- [Batc76] K. E. Batcher, "The FLIP Network in STARAN," *Proceedings of the 1976 International Conference on Parallel Processing*, pp. 65-71 (August 1976).
- [Bhar83] K. Bharath-Kumar and J. M. Jaffe, "Routing to Multiple Destinations in Computer Networks," *IEEE Transactions on Communications COM-31*(3) pp. 343-351 (March 1983).
- [Brow80] S. Browning, *Communications in a Tree Machine*, Bell Laboratories internal memorandum (Jan. 1980).
- [Brui46] D. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Academie van Wetenschappen te Amsterdam, Proc. Section of Sciences* 49(7) pp. 758-764 (1946).
- [Burr71] C. Burrus, "Block Implementation of Digital Filters," *IEEE Transactions on Circuit Theory CT-18* pp. 697-701 (Nov. 1971).
- [Burr72] C. Burrus, "Block Realization of Digital Filters," *IEEE Transactions on Audio and Electroacoustics AU-20* pp. 230-235 (Oct. 1972).
- [Cant74] D. G. Cantor and M. Gerla, "Optimal Routing in a Packet-Switched Computer Network," *IEEE Transactions on Computers C-23*(10) pp. 1062-1069 (Oct. 1974).
- [Carr72] W. Carr and J. Mize, *MOS/LSI Design and Applications*, Texas Instruments Inc. (1972).
- [Chen81] P. Chen, D. Lawrie, D. Padua, and P. Yew, "Interconnection Networks Using Shuffles," *Computer* 14(12) pp. 55-64 (Dec. 1981).
- [Chou81] W. Chou, A. W. Bragg, and A. A. Nilsson, "The Need for Adaptive Routing in the Chaotic and Unbalanced Traffic Environment," *IEEE Transactions on Communications COM-29*(4) pp. 481-490 (April 1981).
- [Chua75] L. Chua and P. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).
- [Chu77] W. W. Chu and M. Y. Shen, *A Hierarchical Routing and Flow Control Policy (HRFC) for Packet Switched Networks*, North Holland Publishing Co., Amsterdam (1977).
- [Clos53] C. Clos, "A Study of Nonblocking Switching Networks," *Bell System Technical Journal* 32 pp. 406-424 (1953).
- [Dahl74] G. Dahlquist, A. Bjorck, and N. Anderson, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).

- [Dala78] Y. K. Dalal and R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM* 21(12) pp. 1040-1048 (December 1978).
- [Davi73] D. W. Davies and D. L. A. Barber, *Communication Networks for Computers*, John Wiley & Sons, National Physical Laboratory, Teddington, England (1973).
- [Deke83] E. Dekel and S. Sahni, "Binary Trees and Parallel Scheduling Algorithms," *IEEE Transactions on Computers* C-32(3) pp. 307-315 (March 1983).
- [Desp78] A. Despain and D. Patterson, "X-Tree: A Tree Structured Multiprocessor Computer Architecture," *Proceedings of the 5th Annual Symposium on Computer Architecture, Palo Alto, Ca.* 6(7) pp. 144-151 (April 1978).
- [Dias81a] D. Dias and J. Jump, "Packet switching Interconnection Networks for Modular Systems," *Computer* 14(12) pp. 43-53 (Dec. 1981).
- [Dias81b] D. M. Dias and J. R. Jump, "Analysis and Simulation of Buffered Delta Networks," *IEEE Transactions on Computers* C-30(4) pp. 273-282 (April 1981).
- [Ensl74a] P.H. Enslow, "Appendices I & J: IBM System 360 & 370," pp. 238-256 in *Multiprocessors and Parallel Processing*, John Wiley & Sons (1974).
- [Ensl74b] P.H. Enslow, "Appendices N, O, & P: UNIVAC 1108, 1110, and AN/UYK-7," pp. 290-327 in *Multiprocessors and Parallel Processing*, John Wiley & Sons (1974).
- [Farh81] G. Farhi, "Diametres dans les graphes et numerotations graceuses," Ph. D. Thesis, l'Universite de Paris Sud (April 1981).
- [Feng81] T. Feng, "A Survey of Interconnection Networks," *Computer* 14(12) pp. 12-27 (Dec. 1981).
- [Fink80] R. A. Finkel and M. H. Solomon, "Processor Interconnection Strategies," *IEEE Transactions on Computers* C-29(5) pp. 360-371 (May 1980).
- [Fink81] R. A. Finkel and M. H. Solomon, "The Lens Interconnection Strategy," *IEEE Transactions on Computers* C-30(12) pp. 960-965 (Dec. 1981).
- [Floy62] R. W. Floyd, "Algorithm 97: Shortest Paths," *Communications of the ACM* 5(6) p. 345 (June 1962).

- [Fran71] H. Frank and W. Chou, "Routing in Computer Networks," *Networks* 1(2) pp. 99-112 (1971).
- [Fran81] M. A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communication Networks," *IEEE Transactions on Computers* C-30(4) pp. 283-291 (April 1981).
- [Fran82] M. A. Franklin, D. F. Wann, and W. J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Networks," *IEEE Transactions on Computers* C-31(11) pp. 1109-1116 (Nov. 1982).
- [Frie66] H. D. Friedman, "A Design for (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-15 pp. 253-254 (April 1966).
- [Fuji80] R. M. Fujimoto, "Routing Controller for X-Tree," Master's Report, UC Berkeley (Dec. 1980).
- [Fuji83] R. M. Fujimoto, "Simon: A Simulator of Multicomputer Networks," ERL Report, in preparation (1983).
- [Gall77] R. G. Gallager, "A Minimum Delay Routing Algorithm using Distributed Computation," *IEEE Transactions on Communications* COM-25(1) pp. 73-85 (Jan. 1977).
- [Gerl81] M. Gerla, "Routing and Flow Control," pp. 122-174 in *Protocols and Techniques for Data Communication Networks*, ed. F. F. Kuo, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).
- [Ghee82] T. Gheewala, "The Josephson Technology," *Proc. IEEE* 70(1) pp. 26-34 (Jan. 1982).
- [Glas78] A. Glaser and G. Subak-Sharpe, "Failure, Reliability and Yield of Integrated Circuits," pp. 746-799 in *Integrated Circuit Engineering*, Addison-Wesley, Reading, MA (1978).
- [Goke73] L. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proceedings of the 1st Annual Symposium on Computer Architecture, Gainesville, Florida* 2(4) pp. 21-28 (Dec. 1973).
- [Good81] J. R. Goodman and C. H. Séquin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Transactions on Computers* C-30(12) pp. 923-933 (Dec. 1981).
- [Gott82] A. Gottlieb and J. T. Schwartz, "Networks and Algorithms for Very-Large-Scale Parallel Computation," *Computer, Special Issue on Highly Parallel Computing* 15(1) pp. 27-36 (Jan. 1982).
- [Grif79] M. Griffin, "X-Tree Communication Buses," Master's Report, UC Berkeley (August 1979).

- [Hear70] F. Heart, R. Kahn, S. Ornstein, W. Crother, and D. Walden, "The Interface Message Processor for the ARPA Computer Network," *Proceedings AFIPS Spring Joint Computer Conference* 36 pp. 551-567 (May 1970).
- [Hodg80] C. J. M. Hodges, T. P. Barnwell III, and D. McWhorter, "The Implementation of an all Digital Speech Synthesizer Using a Multimicroprocessor Architecture," *1980 Intl. Conf. on ASSP, Denver, Colorado*, pp. 855-858 (April 1980).
- [Hopp79] A. Hopper and D. Wheeler, "Binary Routing Networks," *IEEE Transactions on Computers* C-28(10) pp. 699-703 (Oct. 1979).
- [Horo81] E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI," *IEEE Transactions on Computers* C-30(4) pp. 247-253 (April 1981).
- [Hosh83] T. Hoshino, T. Kawai, T. Shirakawa, J. Higashino, A. Yamaoka, H. Ito, T. Sato, and K. Sawada, "PACS: A Parallel Microprocessor Array for Scientific Calculations," *ACM Transactions on Computer Systems* 1(3) pp. 195-221 (August 1983).
- [Imas81] M. Imase and M. Itoh, "Design to Minimize Diameter in Building Block Networks," *IEEE Transactions on Computers* C-30(6) pp. 439-442 (June 1981).
- [Inst76] Engineering Staff of Texas Instruments, *The TTL Data Book for Design Engineers - Second Edition*, Texas Instruments Inc., Dallas, Texas (1976).
- [Irla78] M. I. Irland, "Buffer Management in a Packet Switch," *IEEE Transactions on Communications* COM-26(3) pp. 328-337 (March 1978).
- [Jack57] J. Jackson, "Networks of Waiting Lines," *Operations Research* 5(4) pp. 518-521 (August 1957).
- [Jans80] P. Jansen and J. Kessels, "The DIMOND: A Component for the Modular Construction of Switching Networks," *IEEE Transactions on Computers* C-29(10) pp. 884-889 (Oct. 1980).
- [Joel79] A. E. Joel Jr., "Circuit Switching: Unique Architecture and Applications," *Computer* 12(6) pp. 10-22 (June 1979).
- [Juen76] R. R. Jueneman and G. S. Kerr, "Explicit Path Routing in Communications Networks," *Proceedings from International Conference on Computer Communications*, pp. 340-342 (August 1976). Toronto
- [Kahn72] R. E. Kahn and W. R. Crowther, "Flow Control in a Resource-Sharing Network," *IEEE Transactions on Communications* COM-20(3) pp. 539-546 (June 1972).

- [Kamo76] F. Kamoun, "Design Considerations for Large Computer Communications Networks," Ph. D. Dissertation, Engineering Report 7642, UCLA, Los Angeles, California (April 1976).
- [Kerm79] P. Kermani and L. Kleinrock, "Virtual Cut Through: A New Computer Communication Switching Technique," *Computer Networks* 3(4) pp. 267-296 (Sept. 1979).
- [Keye79] R. Keyes, "The Evolution of Digital Electronics Towards VLSI," *IEEE Journal of Solid-State Circuits* SC-14(4) pp. 193-201 (April 1979).
- [Klei75] L. Kleinrock, *Queueing Systems, Volume I: Theory*, John Wiley & Sons (1975).
- [Klei76] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, John Wiley & Sons (1976).
- [Knut73] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching* (vol. 3), Addison Wesley, Reading, Massachusetts (1973).
- [Korn67] I. Korn, "On (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-16 p. 90 (Feb. 1967).
- [Kung80] H.T. Kung and C.E. Leiserson, "Algorithms for VLSI Processor Arrays," pp. 271-292 in *Introduction to VLSI Systems*, ed. C.A. Mead and L.A. Conway, Addison-Wesley, Reading, MA (1980).
- [Kung82] H. T. Kung, "Why Systolic Architectures," *Computer, Special Issue on Highly Parallel Computing* 15(1) pp. 37-46 (Jan. 1982).
- [Kuo81] F. F. Kuo, *Protocols and Techniques for Data Communication Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).
- [Kush82] T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image Processing on ZMOB," *IEEE Transactions on Computers* C-31(10) pp. 943-951 (Oct. 1982).
- [Laur79] M. Laurent, "Input-Output Ports for the X-Tree Nodes," Master's Report, UC Berkeley (Nov. 1979).
- [Lawr75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers* C-24(12) pp. 1145-1155 (Dec. 1975).
- [Lela82a] W. E. Leland and M. H. Solomon, "Dense Trivalent Graphs for Processor Interconnection," *IEEE Transactions on Computers* C-31(3) pp. 219-222 (March 1982).
- [Lela82b] W. E. Leland, "Density and Reliability of Interconnection Topologies for Multicomputers," Ph. D. Thesis, Computer Sciences Technical Report #478, University of Wisconsin, Madison (July 1982).

- [Lev81] G. F. Lev, N. Pippenger, and L. G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Transactions on Computers* C-30(2) pp. 93-100 (Feb. 1981).
- [Long80] S.I. Long, F.S. Lee, R. Zucca, B.M. Welch, and R.C. Eden, "MSI High-speed Low-power GaAs Integrated Circuits using Schottky Diode FET Logic," *IEEE Trans. Microwave Theory Tech.* MTT-28(5) pp. 466-472 (May 1980).
- [Lu83] H. Lu, "High Speed IIR Digital Filters," Ph. D. Dissertation, in preparation (1983).
- [Mass79] G. M. Masson, G. C. Ginger, and S. Nakamura, "A Sampler of Circuit Switched Networks," *Computer* 12(6) pp. 32-48 (June 1979).
- [McQu74] J. McQuillan, "Adaptive Routing Algorithms for Distributed Computer Networks," NTIS Report (AD-781 467), U. S. Department of Commerce (May 1974).
- [McQu78] J. M. McQuillan, "Enhanced Message Addressing Capabilities for Computer Networks," *Proceedings of the IEEE* 66(11) pp. 1517-1527 (Nov. 1978).
- [McQu80] J. M. McQuillan, I. Richer, and E. C. Rosen, "The New Routing Algorithm for the Arpanet," *IEEE Transactions on Communications* COM-28(5) pp. 711-719 (May 1980).
- [Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980). Gov't. ordering no. AD-781 467
- [Memm82] G. Memmi and Y. Raillard, "Some New Results About the (d,k) Graph Problem," *IEEE Transactions on Computers* C-31(8) pp. 784-791 (August 1982).
- [Merl79] P. M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol," *IEEE Transactions on Communications* COM-27(9) pp. 1280-1288 (Sept. 1979).
- [Merl80a] P. M. Merlin and P. J. Schweitzer, "Deadlock Avoidance - Store and Forward Deadlock," *IEEE Transactions on Communications* COM-28(3) pp. 345-354 (March 1980).
- [Merl80b] P. M. Merlin and P. J. Schweitzer, "Deadlock Avoidance in Store and Forward Networks II - Other Deadlock Types," *IEEE Transactions on Communications* COM-28(3) pp. 355-380 (March 1980).
- [Mitr78] S. Mitra and R. Gnanasekaran, "Block Implementation of Recursive Digital Filters - New Structures and Properties," *IEEE Transactions on Circuits and Systems* CAS-25 pp. 200-207 (April 1978).



- [Nage75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Ph. D. Thesis (ERL-M520), University of California, Berkeley (1975).
- [Nass79] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Transactions on Computers* C-27(1) pp. 2-7 (Jan. 1979).
- [Nass81] D. Nassimi and S. Sahni, "A Self-Routing Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers* C-30(5) pp. 332-340 (May 1981).
- [Nass82] D. Nassimi and S. Sahni, "Parallel Algorithms to Set Up the Benes Permutation Network," *IEEE Transactions on Computers* C-31(2) pp. 148-154 (Feb. 1982).
- [Nath83] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt, "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees," *IEEE Transactions on Computers* C-32(6) pp. 569-581 (June 1983).
- [Park80] D. S. Parker, "Notes on Shuffle/Exchange Type Networks," *IEEE Transactions on Computers* C-29(3) pp. 213-222 (March 1980).
- [Pate81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers* C-30(10) pp. 771-780 (Oct. 1981).
- [Patt80] D. A. Patterson and C. H. Séquin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers* C-29(2) pp. 108-116 (February 1980).
- [Patt82] D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer* 15(9) pp. 8-21 (Sept. 1982).
- [Peas77] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Transactions on Computers* C-26(5) pp. 548-573 (May 1977).
- [Pouz76] L. Pouzin, "Flow Control in Data Networks - Methods and Tools," *Proceedings of the Third International Conference on Computer Communications, Toronto, Canada*, pp. 467-474 (August 1976).
- [Pouz78] L. Pouzin and H. Zimmerman, "A Tutorial on Protocols," *Proceedings of the IEEE* 66(11) pp. 1346-1370 (Nov. 1978).
- [Pouz81] L. Pouzin, "Methods, Tools, and Observations on Flow Control in Packet-Switched Data Networks," *IEEE Transactions on Communications* COM-29(4) pp. 413-426 (April 1981).
- [Prad80] D. K. Pradhan and K. L. Kodandapani, "A Uniform Representation of Single- and Multistage Interconnection Networks Used in SIMD



- Machines," *IEEE Transactions on Computers* C-29(9) pp. 777-791 (Sept. 1980).
- [Prad82] D.K. Pradhan and S.M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," *IEEE Trans. on Computers* C-31(9) pp. 863-870 (Sept. 1982).
- [Prep81] F. P. Preparata and J. Vuillemin, "The Cube Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM* 24(5) pp. 300-309 (May 1981).
- [Prep83] F. P. Preparata, "A Mesh-Connected Area-Time Optimal VLSI Multiplier of Large Integers," *IEEE Transactions on Computers* C-32(2) pp. 194-198 (Feb. 1983).
- [Reed83] D. A. Reed and H. D. Schwetman, "Cost-Performance Bounds for Multicomputer Networks," *IEEE Transactions on Computers* C-32(1) pp. 83-95 (Jan. 1983).
- [Rile82] D. D. Riley and R. J. Baron, "Design and Evaluation of a Synchronous Triangular Interconnection Scheme for Interprocessor Communications," *IEEE Transactions on Computers* C-31(2) pp. 110-118 (Feb. 1982).
- [Rind77] J. Rinde, "Routing and Control in a Centrally Directed Network," *AFIPS Conference Proceedings, National Computer Conference* 46 pp. 803-808 (1977).
- [Sega77] A. Segall, "The Modelling of Adaptive Routing in Data-Communication Networks," *IEEE Transactions on Communications* COM-25(1) pp. 85-95 (Jan. 1977).
- [Sega81] A. Segall, "Advances in Verifiable Fail-Safe Routing Procedures," *IEEE Transactions on Communications* COM-29(4) pp. 491-497 (April 1981).
- [Sequ78] C. H. Séquin, A. Despain, and D. Patterson, "Communications in X-Tree, A modular Multiprocessor System," *Conference Proceedings, ACM*, (Dec. 1978).
- [Sequ82] C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," in *Proc. Advanced Course on VLSI Architecture*, Univ. of Bristol, England, ed. P. Treleaven, Prentice Hall, Englewood Cliffs, New Jersey (1982).
- [Shoc80] J. Shoch and J. Hupp, "Measured Performance of an Ethernet Local Network," *Communications of the ACM* 23(10) pp. 711-721 (Dec. 1980).
- [Sieg79a] H. Siegel, "Interconnection Networks for SIMD Machines," *Computer* 12(6) pp. 57-65 (June 1979).

- [Sieg81] H. Siegel and R. McMillen, "The Multistage Cube: A Versatile Interconnection Network," *Computer* 14(12) pp. 65-76 (Dec. 1981).
- [Sieg79b] H. J. Siegel, "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Transactions on Computers* C-28(12) pp. 907-917 (Dec. 1979).
- [Spro81] D. Sproule and F. Mellor, "Routing, Flow and Congestion Control in the Datapac Network," *IEEE Transactions on Communications* COM-29(4) pp. 386-391 (April 1981).
- [Ste183] D. Steinberg, "Invariant Properties of the Shuffle-Exchange and a Simplified Cost-Effective Version of the Omega Network," *IEEE Transactions on Computers* C-32(5) pp. 444-450 (May 1983).
- [Ston71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers* C-20(2) pp. 153-161 (Feb. 1971).
- [Ston72] H. S. Stone, "Dynamic Memories with Enhanced Data Access," *IEEE Transactions on Computers* C-21(4) pp. 359-366 (April 1972).
- [Stor70] R. M. Storwick, "Improved Construction Techniques for (d,k) Graphs," *IEEE Transactions on Computers* C-19 pp. 1214-1216 (Dec. 1970).
- [Stri83] L. Stringa, "EMMA: An Industrial Experience on Large Multiprocessing Architectures," *Proceedings of the 10th Annual Symposium on Computer Architecture, Stockholm, Sweden* 11(3) pp. 328-333 (June 1983).
- [Swan77a] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "CM\*-A Modular, Multi-microprocessor," *Proceedings of the National Computer Conference, Dallas, Texas*, pp. 637-643 (June 1977).
- [Swan77b] R. J. Swan, A. Bechtolsheim, K. Lai, and J. K. Ousterhout, "The Implementation of the CM\* Multi-microprocessor," *Proceedings of the National Computer Conference, Dallas, Texas*, pp. 645-655 (June 1977).
- [Swar82] E. Swartzlander Jr. and B. Gilbert, "Supersystems: Technology and Architecture," *IEEE Transactions on Computers* C-31(5) pp. 399-409 (May 1982).
- [Taj77] W. Tajibnapis, "A Correctness Proof of a Topology Maintenance Protocol for a Distributed Computer Network," *Communications of the ACM* 20(7) pp. 477-485 (July 1977).
- [Tane81] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

- [Than81] S. Thanawstien and V. P. Nelson, "Interference Analysis of Shuffle/Exchange Networks," *IEEE Transactions on Computers C-30*(8) pp. 545-556 (August 1981).
- [Thom80] C. D. Thompson, "A Complexity Theory for VLSI," Ph. D. Dissertation, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa. (August 1980).
- [Toue79] S. Toueg and K. Steiglitz, "The Design of Small-Diameter Networks by Local Search," *IEEE Transactions on Computers C-28*(7) pp. 537-542 (July 1979).
- [Tyme81] L. Tymes, "Routing and Flow Control in TYMNET," *IEEE Transactions on Communications COM-29*(4) pp. 392-398 (April 1981).
- [Wagn83] R. A. Wagner, "The Boolean Vector Machine (BVM)," *Proceedings of the 10th Annual Symposium on Computer Architecture, Stockholm, Sweden* 11(3) pp. 59-66 (June 1983).
- [Widd80] L.C. Widdoes, "The S-1 Project: Developing High-Performance Digital Computers," *Proc. COMPCON*, pp. 282-291 (Feb. 1980).
- [Wing80] O. Wing and J. W. Huang, "A Computational Model of Parallel Solution of Linear Equations," *IEEE Transactions on Computers C-29*(7) pp. 632-638 (July 1980).
- [Witt81] L. D. Wittie, "Communications Structures for Large Networks of Microcomputers," *IEEE Transactions on Computers C-30*(4) pp. 264-273 (April 1981).
- [Wong81] R. Wong, "Serial Communications in X-Tree," Master's Report, UC Berkeley (June 1981).
- [Wu80a] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers C-29*(8) pp. 694-702 (August 1980).
- [Wu80b] C. Wu and T. Feng, "The Reverse-Exchange Interconnection Network," *IEEE Transactions on Computers C-29*(9) pp. 801-811 (Sept. 1980).
- [Wu81a] C. Wu and T. Feng, "The Universality of the Shuffle-Exchange Network," *IEEE Transactions on Computers C-30*(5) p. 324 (May 1981).
- [Wulf72] W. A. Wulf and C. G. Bell, "C.MMP--A Multi Mini Processor," *Proceedings of the AFIPS Fall Joint Computer Conference, Montvale, N. J.* 41 pp. 765-777 (1972).
- [Wu81b] S. B. Wu and M. T. Liu, "A Cluster Structure as an Interconnection Network for Large Multimicrocomputer Systems," *IEEE*

*Transactions on Computers C-30*(4) pp. 254-264 (April 1981).

- [Yew81] P. Yew and D. H. Lawrie, "An Easily Controlled Network for Frequently Used Permutations," *IEEE Transactions on Computers C-30*(4) p. 296 (April 1981).
- [Yu84] W. Yu, "LU Decomposition on a Multiprocessing System with Communication Delay," Ph. D. Dissertation, in preparation (1984).
- [Zema81] J. Zeman and A. Lindgren, "Fast Digital Filters with Low Round-Off Noise," *IEEE Transactions on Circuit and Systems CAS-28* pp. 716-723 (July 1981).
- [Zimm80] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications COM-28*(4) pp. 425-432 (April 1980).

# SIMON: A SIMULATOR OF MULTICOMPUTER NETWORKS

*Richard M. Fujimoto*

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California, Berkeley, CA 94720  
August 31, 1983

## ABSTRACT

This document describes a simulator for modelling execution of parallel programs on a multiprocessor. The simulator executes a set of programs as if each were run on a separate processor, and compiles statistics for the entire run. A portion of the simulator, known as the "switch model", simulates the exchange of messages among the processors through an interconnection network. The simulator was developed to allow different types of interconnection hardware (e. g. packet switches, crossbars, etc.) to be modelled by simply "plugging in" the appropriate switch model.

Applications are programmed as a set of communicating tasks (processes). The interface seen by the applications programmer is discussed, and in particular, the communications mechanism is described in detail. Examples are given. Finally, the implementation of Simon is described.

## 1. INTRODUCTION

This document describes a simulation program called Simon (simulator of multicoputer networks). It is assumed that the reader has at least a reading knowledge of the C programming language [Kern79]. All applications programs written for Simon should be written in C.

### 1.1. What is Simon?

Simon is a deterministic event driven simulation program which models the execution of a parallel application program on a multiprocessor computing system. The application program, provided by the user, consists of a number of sequential programs which execute concurrently. Simon executes each sequential program as if it were run on a separate processor. Message transmissions between the processors are simulated, and statistics concerning the program's dynamic behavior (e.g. execution time, time spent waiting for data, etc.) are reported.

Simon consists of three distinct modules:

- (1) The application program.
- (2) The simulator base.
- (3) The switch model.

Each of these will now be described in turn.

The first component, the application program, consists of a number of user defined "tasks" written in C. A task is defined as a sequential program, and the data it uses, executing on a processor. The tasks execute asynchronously, and communicate by exchanging messages. Intertask communication is made possible by routines provided by Simon for sending and receiving messages. No shared memory is allowed, i.e. no two tasks can directly access the same memory location. A task is conceptually identical to a process, however this term is reserved to refer to a Simon program executing on a host computer.

The second component, the simulator base, time multiplexes execution of the tasks on the host computer, in this case a VAX. The base also collects statistics on the application program and outputs them when execution completes.

The third component, the switch model, simulates the transmission of messages among the processors. The switch model might be, to mention a few, a crossbar, an Ethernet, or a store-and-forward communications network. By separating the model for the interconnection switch into a separate module, alternate switching structures can be compared by just "plugging in" the appropriate switch models.

### 1.2. Who Should Use Simon?

Simon was written with two types of users in mind:

- (1) Persons interested in developing and analyzing parallel application programs.
- (2) Persons interested in analyzing the performance of various interconnection schemes under loads generated by real application programs.

In the first case, applications such as circuit simulation and voice recognition are envisioned, although Simon is by no means restricted to these applications. In the second case, Simon provides an alternative to simulation models based on loads created artificially by random number generators.

### 1.3. Restrictions

In order to reduce the complexity and overhead of running Simon, several restrictions have been made. First, this implementation assumes that each task runs on its own processor. Thus, two distinct task may not execute on the same processor. The maximum number of processors (tasks) however, is virtually unlimited, subject only to memory constraints on the computer running Simon. Further, it is assumed that tasks and communications among tasks are statically defined, i. e. all tasks must be created before any can begin execution, and each task must initially specify all other tasks with which it may send or receive messages.



#### 1.4. About this Document

This document is intended for those who are interested in writing parallel application programs for Simon, or for those interested in understanding its internal structure and implementation. For the former, mechanisms for creating tasks are defined, as well as intertask communications facilities. For the latter, an overview of the current implementation of Simon for a VAX-11/780 will be discussed.

The remainder of this document consists of six sections, and a number of appendices. The next two sections describe what the user needs to know to write application programs. Section 2 describes the routines provided for creating and executing tasks, and section 3 describes routines for exchanging messages. Section 4 gives some examples using the routines defined in the previous two sections. Section 5 describes the timing model which is used for gathering statistics on the task. Section 6 gives some guidelines for writing programs for Simon. These guidelines are provided to avoid some rather subtle bugs in applications programs which Simon cannot detect. Finally, section 7 describes the internal organization of the Simon program.

A summary of all of the routines discussed in this document is given in Appendix I. Other appendices describe the mechanics of using Simon and the special C compiler necessary for the timing model.

## 2. TASKS

An application program can view SIMON as a set of subroutines for performing various functions, much like a C program can view an operating system as a set of subroutines for, e. g., printing results or reading data from a file. This section and the next describe the functions provided by SIMON as well as the routines which implement them.

The application program executing on the multiprocessor is a set of concurrently executing "tasks" (or equivalently, processes). A task can be thought of as an instance (i. e. a loaded and executing core image) of a C program. A program, like that in a conventional uniprocessor, consists of a set of routines which call each other in some arbitrary fashion. One routine in the program for a task acts as the "main procedure", and is called when the task first begins execution. This routine should *not* be called "main".

Several tasks may be created from a single program. To the user, this is equivalent to generating multiple copies of the program, and creating a single task for each one. In reality however, SIMON shares a single copy of the code among the tasks, and creates a separate data area for each.

This section describes the routines provided by SIMON for creating and executing tasks. First, task creation via the "mktask()" routine is described. A task cannot be created however, unless there already exists another task to create it, presenting a "chicken and egg" problem. To solve this problem, a routine called "bootstrap()" is provided for getting the simulator started. This is discussed in the following subsection. If multiple tasks are created from a single program and that program uses static or global variables, a routine, called "init()", must be used. This is the subject of the third subsection.

### 2.1. Creating Tasks: The Mktask() Routine

SIMON provides a routine called mktask() for creating tasks. Mktask() performs several functions. In addition to "creating" the task, mktask() assigns the task a user provided name and task "id". Every task has a unique id which is used internally to distinguish it from other tasks (note that task names need not

be unique). This id may also be used in conjunction with some switch models to assign certain tasks to execute on specific processors. Space for maintaining information necessary for the task to execute (e. g. state information) is also allocated, as well as space for collecting statistics. Mktask() is also used to identify the task's main procedure. When the task first begins execution, this procedure is called via an ordinary subroutine call. Finally, mktask() allows the user to specify parameters which are passed to the created task when it begins execution.

Mktask() takes five parameters, and is defined as follows:

**mktask (name, cdptr, id, parm, lngth):**

where:

- name** = a character string specifying the name of the task.
- cdptr** = a pointer to the main procedure for the task.
- id** = a positive integer specifying the id to be assigned to the task.
- parm** = a pointer to a parameter list.
- lngth** = the length of this parameter list in bytes.

It is an error to assign two tasks to the same id, or to assign a task to id 0. It will be seen later that task id 0 is reserved for the "bootstrap" task.

The last two parameters are used to copy the parameter list into an internal buffer. A pointer to this buffer is passed to the task when it is first called. If the size of the parameter list is specified as zero, mktask() assumes no parameters are to be passed to the task. The parameter pointer must be NULL in this case, or an error will result.

Thus, the main procedure of each task must have at most one parameter, a pointer to parameter(s) used by that task. If a task requires more than one parameter, a structure should be created, and a pointer to this structure passed to mktask().

The parameter list passed to a task should not contain pointers, i. e. all parameters passed to the task should physically reside in the block of memory specified by the last two parameters passed to mktask(). Mktask() copies the

specified data without interpretation, so any data stored at the address specified by the pointer may have been changed by the time the task begins to execute. Passing pointers in the parameter list can lead to rather obscure bugs. An example of this will be seen later.

To illustrate the use of `mktask()`, a few examples will now be given. The simplest form of `mktask()` is shown below:

```
bootstrap()
{
    int foo();
    mktask ("foo", foo, 5, NULL, 0);
}

foo()
{
    code for task foo
}
```

This creates a task called "foo" which is assigned to task id 5. The main procedure of the task is also called foo. Foo is not passed any parameters. Bootstrap() is a task already in existence, and will be discussed in the next subsection.

A second example demonstrating the passing of a single parameter to a task is given below:

```
bootstrap()
{
    int fle();
    int i;

    i = 20;
    mktask ("fle", fle, 6, &i, sizeof (int));
}

fle (p)
int *p;
{
    printf ("Parameter passed is %d\n", *p);
}
```

Here, task fle is created and assigned to task id 6. When executed, the line "Parameter passed is 20" will be displayed on the terminal screen. Note the use of the `sizeof()` routine to get the size, in bytes, of a C object.

Finally, a third example demonstrates the passing of several parameters to two different task via structures.

```
struct parm /* format of parameters */
{
int a[10]; /* Pass an array */
int n; /* number of elements */
};

bootstrap()
{
int T();
struct parm p;
int i, id;

id = 1; /* id assigned to tasks */

p.n = 10;
for (i=0; i<p.n; i++) p.a[i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));

for (i=0; i<p.n; i++) p.a[p.n-1-i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));
}

T(p)
struct parm *p;
{
int i;

for (i=0; i<p->n; i++)
    printf ("%d\n", p->a[i]);
}
```

Two tasks are created from the program T. The first is passed an array of integers in ascending order, and the second is passed an array in descending order. Note that the structure parm contains no pointers. Suppose we decided to pass the tasks a pointer to the data rather than the data itself. Bootstrap() might then be defined as follows:

```
struct parm /* format of parameters */
{
int *a; /* Pass a pointer!! */
int n; /* number of elements */
};

bootstrap()
{
```

```
int T();
struct parm p;
int i, id, buf[10];

id = 1;
p.n = 10;
p.a = buf;

for (i=0; i<p.n; i++) p.a[i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));

for (i=0; i<p.n; i++) p.a[p.n-1-i] = i;
mktask ("T", T, id++, &p, sizeof (struct parm));
}
```

There are two reasons why this program malfunctions. First, buf is declared as an automatic (i.e. non-static local) variable, and is thus kept on the runtime stack for bootstrap(). The pointer passed to the tasks is thus a pointer into the runtime stack! When bootstrap() returns, its stack is returned to the system, and subsequently overwritten by other procedures. Thus, the created tasks receive garbled data. However, even if buf were declared as a global variable (and thus, not kept on the stack), there is still another problem. The first task is passed only a pointer to the data, and not the data itself. The second "for loop" in bootstrap() modifies this data, and so the result is *both* tasks receive the array in descending order. In the previous (correct) example, the actual data was passed via mktask() to the newly created task. Since mktask() creates a separate copy of this data, it cannot be changed by any user program. For this reason, it is recommended that structures passed to tasks as parameters contain no pointers fields.

## 2.2. Getting Started: The Bootstrap() Routine

In order for the user to create and execute tasks, he must execute mktask(). In order to execute mktask(), he must have created and executed a task. To end this cycle, SIMON assumes the preexistence of a user-defined task called bootstrap(). Bootstrap() is passed no parameters, and is arbitrarily assigned to task id 0.

Bootstrap() uses mktask() to create the user tasks, and then returns control back to the simulator, never to be executed again. It's sole purpose is to

create the set of tasks the user wishes to execute. Bootstrap() is not allowed to communicate with any other task. All tasks, including bootstrap(), are created and begin execution, at time 0, regardless of when mktask() was executed. Thus, in this respect, bootstrap is a somewhat artificial task used only to get the simulation started. Within SIMON, bootstrap() is treated as any ordinary task however.

The first implementation of SIMON assumes that tasks cannot be created dynamically. All tasks must be created by bootstrap(). Thus, it is an error for any task other than bootstrap() to execute the mktask() routine.

### **2.3. Multiple Instances of Tasks: The Init() Routine**

In sequential programming, one often writes a subroutine to perform some function independently of the specific data it is to operate on. This function is then called from various points in the program, each time specifying the data via parameters. Thus, the algorithm remains the same, but the data varies from call to call. Each call to the subroutine creates an "instance" of that subroutine. Local variables are created, actual parameters (those specified in the function call) are mapped to formal parameters (those specified in the subroutine), etc. Since execution is sequential, at most one instance of the subroutine exists at any given time, ignoring recursion.

Similarly, it is useful to write one program and then create many tasks from this program which only differ in the data being operated on. Here however, the various tasks execute concurrently, and thus exist simultaneously. Just as one parameterizes instances of a subroutine, one must also parameterize tasks (as described above in the mktask() routine). This is useful for example, in array operations where each task performs the same computation, but on different parts of the array.

When multiple instances of a task are created from a program using non-automatic (i. e. static or global) variables, the simulator defined routine init() must be used. Init() tells SIMON where these variables reside. SIMON must have this information so that these variables can be saved and restored when



execution switches among tasks created from the same program. Automatic variables are stored on the runtime stack of the executing program, and are saved and restored by SIMON when necessary, transparent to the user.

Non-automatic variables should all be stored in a single block of memory which will be referred to here as the "global data area". `Init()` has two parameters, and is defined as follows:

`init (glob, lngth);`

where:

`glob` = a pointer to the start of the global data area.

`lngth` = the length of this area in bytes.

`Init()` need only be used in a task if multiple instances are created. It must be executed exactly once by such tasks, and must execute before any other simulator defined routine is executed.

To use `init()`, the user should:

- (1) place ALL of the non-automatic variables for the task into a structure.
- (2) call `init()` using a pointer to this structure, and a size gotten from `sizeof()`.

In addition, it is recommended that arrays be created dynamically via one of the storage allocation routines (e. g. `calloc()`) provided in C. This will reduce the execution time of the simulator, since variables created by (say) `calloc()` need not be saved and restored when execution switches from one task to another. This is because each task (even those created from the same program) dynamically creates it's own data area to which it has sole access. In other instances, this recommendation becomes a requirement, since each task is allocated a fixed size area for saving its stack. If an attempt is made to save more data than this area can hold, an execution error results. Data areas created by `calloc()` are not kept on the stack.

In the discussion above, it is assumed that non-automatic variables are used to share data between procedures of the same task. Different tasks must not use global variables to exchange information, as this compromises the simulation by performing communications between tasks without using the switch

model.

The following is an example of a program from which several tasks are created. Note the use of `calloc()` for creating arrays, and the manner in which non-automatic variables are declared so their location can be passed to SIMON via `init()`.

```
static struct gstruct /* Global data area */
{
int a, b, c;          /* Some scalars */
int *g_arr;           /* A global array */
} glob;

foo ()                /* Program for a task */
{
int i, j;             /* Automatic scalar */
int *a_arr;           /* Automatic array */
char *calloc();        /* Storage allocator */

/* Define global data area */
init (&glob, sizeof (struct gstruct));

/* Create automatic variable array */
a_arr = (int *) calloc (100, sizeof (int));

/* Create global array */
glob.g_arr = (int *) calloc (20, sizeof (char));

/* Reference globals and automatic locals */
i = 20; j = 30;        /* Automatic scalar */
a_arr[j] = 30;         /* Automatic array */
glob.a = 20;           /* Global scalar */
glob.g_arr[i] = 5;     /* Global array */
.
.
}
```

## 2.4. Other Routines

Finally, two other routines pertaining to tasks will be mentioned. These are `task()`, which returns a task's id, and `taskname()`, which returns a pointer to the task's name (the character string passed to `mktask()` when the task was created). The character string returned must not be overwritten, as it is SIMON's only copy.

The `task()` routine takes no parameters, and returns an integer value. `Taskname()` takes a single parameter, the id of the task in question, and returns a pointer to a character string. Thus, a task can determine it's own name by calling:

```
taskname (task());
```

The information passed through a task's name or its id may be used for further parameterization of the task.

### 3. INTERTASK COMMUNICATIONS

This section describes the mechanisms for allowing tasks to send/receive messages to/from other tasks. Here, two types of communications mechanisms are defined:

- (1) specification of which tasks communicate with which other tasks.
- (2) sending and receiving messages.

The first mechanism uses "fifo's" whose names are globally known throughout the multiprocessor. Each task has a set of "export" and a set of "import" fifo's for sending and receiving messages respectively. A task sends a message by putting it into one of his globally known export fifo's. A message is received by taking it out of an import fifo. The details of how the data is transported from an export to an import fifo is left up to the switch model, and will not be discussed here.

The next subsection discusses the basic communications mechanism in greater detail. Following this, the subroutines implementing each of the two types of communications mechanisms defined above will be described. A concise definition of the routines defined in this section is made in Appendix I.

#### 3.1. Overview of the Communications Mechanisms

Data may be transmitted to or received from other tasks via fifo's. Each task creates a set of fifo's which act as the interface between it and any tasks it communicates with.

Fifo's hold messages which are being sent to or have arrived from other tasks. Thus, elements (i. e. messages) of the same fifo may vary in size. The contents of messages are not interpreted by the simulator.

Each fifo has a user defined name (an arbitrary character string) which is globally known by all other tasks. Furthermore, fifo's are classified as being either "export" or "import". When a task creates an export fifo of some name (say X) it is given the capability to "export" data (i. e. send messages) to all tasks which have an import fifo called X.

A task may have both an export and an import fifo called X, but it cannot have more than one import or export fifo of the same name. Several tasks may each have an import fifo called X, allowing one to easily broadcast data to several tasks. Similarly, several tasks may have export fifo's of the same name, allowing several tasks to send data to one (or more) import fifo(s). Thus, there are no restrictions on creating fifo's other than creating more than one import/export fifo of the same name within a single task.

Thus, communication paths among tasks are defined by the names of the fifo's created within each task. When a task wants to send a message to another task, it puts the data into one of its export fifo's. The switch model removes the data from the export fifo, replicates it as many times as necessary, and places a copy in each import fifo of the same name as the export fifo the data was originally put in. All of this is transparent to the programmer. The receiver now only needs to get the data out of its import fifo. Two tasks must have a commonly named fifo to exchange data directly.

Sequentiality is preserved between any pair of export and import fifo's of the same name. If two messages are placed in export fifo "X", one after the other, the simulator will guarantee that these messages arrive in all import fifo's named "X" in the same order in which they were sent. In network terminology, every pair of identically named export and import fifo's form a "virtual circuit".

The next subsection describes creation of export and import fifo's. The following subsection describes the routines provided for putting (getting) data into export (from import) fifo's.

### **3.2. Fifo Creation**

This subsection describes the simulator defined routines for creating fifo's. Export() and import() are the basic routines used for creating export and import fifo's respectively. Also, arrays of fifo's can be created using the exparr() and imparr() routines. Two additional routines called cnvrr() and cnvrrsv() are provided to "index" into an array of fifo's.

When a task begins executing for the first time, it must create all of the fifo's it will need for the entire simulation run. No dynamic fifo creation is allowed. After this initial execution period (which ends when the task gives up the cpu to allow other tasks to execute) the task must not attempt to create any new fifo's. Any attempt to do so will be flagged as an error.

### 3.2.1. Export()

The export() function creates a single export fifo. It takes a single parameter, a character string (or more accurately, a pointer to a character string) specifying the name of the fifo. When export() is called, the simulator creates an export fifo of this name, allowing the task to send data to all other tasks which have an import fifo of the same name. Thus, the statement:

```
export ("X");
```

creates an export fifo called X.

### 3.2.2. Import()

Import() is very similar to export(). Import() however, takes two parameters. In addition to the name parameter, it requires a parameter specifying the maximum number of messages the import fifo can hold. If a 0 is specified, the length of the fifo is not bounded (this is the most commonly used case). Fifo's with a maximum length specified are called bounded fifo's, and are useful for applications where a task receiving data is not interested in all the data sent to it, but only the last (say) 3 values.

When import() is called, the simulator creates an import fifo of the specified name. This allows the task to receive data sent by other tasks with export fifo's of the same name. Data arriving into a bounded fifo which is full causes the oldest data (at the front of the fifo) to be discarded, the new data to be placed at the end of the fifo, and a flag (readable by overfl() defined below) to be set. Note that only import fifo's have bounded length. For example,

```
import ("Y", 0);  
import ("Z", 3);
```

creates two import fifo's. The fifo called Y is of unlimited length, but the fifo

called Z will only hold up to 3 messages.

### 3.2.3. `Exparr()`, `Imparr()`, `Cnvarr()`, and `Cnvarrsv()`

These routines are used for creating and accessing arrays of fifo's. An "array of fifo's" is simply a set of fifo's, all of which are either export or import fifo's, which follow a certain naming convention. The convention for naming individual fifo's in an array is to concatenate a "base name" with a subscript enclosed by brackets ([ ]). Thus the name of a fifo which is part of an array looks like a C array variable. Subscripts start at 0.

`Exparr()` creates an array of export, and `imparr()` an array of import fifo's. `Exparr()` takes two parameters, one specifying the base name to be used in naming the fifo's, and the other specifying the length of the array. `Imparr()` takes these two parameters plus a third giving the maximum length of the fifo's, or 0 if their length is not limited. All import fifo's of an array must have the same maximum length. `Exparr()` and `imparr()` use `export()` and `import()` respectively to create fifo's. They could easily be written by the user, but are provided as a convenience and to form a convention for naming individual fifo's in a FIFO array.

For example,

```
exparr ("X", 4);
```

creates four export fifo's (with base name X) called "X[0]", "X[1]", "X[2]", and "X[3]". Note that no blanks are automatically inserted into the fifo name. Remember that fifo's which are members of arrays are identical to ordinary fifo's. The only difference is that they follow this naming convention. A task could create a single fifo corresponding to one element of the array by simply calling `export()` (or `import()`), as for example:

```
export ("X[2]");
```

In practice, one would like to access the *i*th fifo, where *i* is an integer variable. To do this, a mapping function called `cnvarr()` is provided. `Cnvarr()` takes two arguments, a character string specifying a base name, and an integer variable specifying a subscript. Thus,



`cnvrr ("X", 2);`

returns the character string "X[2]". No checking is done to see if the resulting name corresponds to any fifo existing in the system. An error will result however, if an attempt is made to use an invalid fifo name.

The value returned by `cnvrr()` should never be stored into another variable. It should only be used as the argument passed to a simulator defined subroutine (e. g. `put()` or `get()`, defined below). The character string returned is only valid until the next time the task executes `cnvrr()`. This is because each task uses a single buffer for holding character strings generated by `cnvrr()`. `Cnvrr()` always returns a pointer to the start of this buffer. Thus, if two successive calls to `cnvrr()` are made, the result of the second will overwrite the first.

If it is necessary to assign the pointer returned by `cnvrr()` to another variable, the `cnvrrsv()` routine should be used. This is identical to `cnvrr()`, except the storage for the resulting string is allocated by `calloc()`, and thus remains indefinitely. `Calloc()` is not used on calls to `cnvrr()` to conserve storage.

### **3.3. Sending and Receiving Messages**

This subsection describes routines provided by the simulator for sending and receiving messages. In addition, routines for interrogating the status of fifo's are also discussed.

Three functions, `put()`, `puts()`, and `get()` are available for putting data into export fifo's and getting data from import fifo's. A routine called `waitf()` allows a task to wait for data to arrive in any of several import fifo's. Routines called `size()` and `qlength()` are available for determining the size of the next message and the number of messages in an import fifo. Finally, a routine called `overflow()` is used to detect overflow of import fifo's which have a limited length (bounded fifo's). The following is a detailed description of each of these routines.

#### **3.3.1. Put() and Puts()**

These two routines are used to send messages, i. e. put data into an export fifo. The `put()` routine takes three parameters. The first is a character string

specifying the name of the export fifo data is to be put into. The second is a pointer to the first byte of data to be sent. The third is the number of bytes to be sent. When called, a contiguous block of data L bytes long (where L is the third parameter) beginning at the memory location specified by the second parameter is copied into one of the simulator's internal buffers (forming a single message), and added to the export fifo specified by the first parameter. Thus each element of the fifo is a block of data (i. e. a message), which for a given fifo, may vary in length from element to element. Thus, the statement

```
put ("X", &i, sizeof(i));
```

sends a message consisting of the integer (assume i is declared as an integer) which is the current value of the variable i.

If a task declares an export and import fifo of the same name, it is usually not desired that a put() cause the task to send data to itself. Thus, put() only causes data to be sent to other tasks. In some situations however, it may be convenient to allow a task to receive its own messages. For example, suppose a task needs to process data which is sometimes created locally within that task, but other times remotely in some other task. Allowing a task to send data to itself allows the receiver to treat the two cases as one. For this purpose, the function puts() (put to self) is defined. Puts() uses the same parameters as put(). The first parameter should be the name of a fifo which is both an export and an import for that task. When a task executes puts(), it sends data to itself as well as to all other import fifo's of the same name. If a task executes puts() on an export fifo for which it does not also have an import fifo of the same name, an error will result.

### 3.3.2. Get()

Get() takes two parameters, a character string specifying the name of some import fifo created by the task, and a pointer to where the received data is to be stored. When called, the next message of the specified import fifo (in the general case, a block of data of arbitrary length) is removed from the fifo and copied into contiguous memory locations starting at the address specified by

the second parameter. If the specified import fifo is empty when `get()` is called, the task waits for data to arrive. The waiting is of course, transparent to the programmer. Thus, the statement

```
get ('X', &j);
```

might be used to receive the value of `i` sent by the task in the `put()` example above. When the `get()` completes execution, `j` will hold whatever data was sent. Note that the above implies that the receiver knows what type of data is being sent. It is up to the user to ensure that the receiver correctly interprets any data sent to it. If the next message in fifo `X` held (say) a floating point number, the simulator would not complain, but the result would, in general, be unpredictable if the receiver were expecting an integer.

### 3.3.3. Waitf()

`Waitf()` takes two parameters. The first is an array of character strings (i. e. pointers to character strings), each of which is the name of an import fifo created by that task. The second parameter is an integer specifying the length of this array. When executed, the simulator checks each of the specified import fifo's, and if all are empty, the task blocks. When a message arrives in one of these fifo's, the task is restarted at the instruction following the `waitf()`. If one or more of the specified fifo's is not empty when the `waitf()` is executed, then the `waitf()` acts like a no-op.

The `waitf()` function allows a task to wait for data to arrive in any of a set of import fifo's. When a message arrives and is placed into one of these import fifo's, the task must determine which fifo the message arrived in. Clearly, `get()` cannot be used for this purpose, since a `get()` on an empty fifo will cause the task to block. Two routines which can perform this function are `size()` and `qlength()`. These will be discussed next.

### 3.3.4. Size() and Qlength()

The `size()` function takes a single parameter specifying an import fifo created by the task. When called, it returns the size (in bytes) of the next message in this fifo. If the fifo is empty, `size()` returns 0. This routine is useful for

receiving variable length data.

`qlength()` also takes a single parameter specifying the name of an import `fifo`. This routine returns the number of messages currently in that `fifo`. If the `fifo` is empty, `qlength()` returns 0. It, like `size()`, can be used for polling `fifo`'s to see if they have any data (see discussion of the `waitf()` routine defined above). For efficiency reasons however, it is recommended that the user use `size()` for this purpose.

### 3.3.5. `Overflow()`

`Overflow()` takes a single parameter specifying an import `fifo` created by the task. Each import `fifo` has a flag associated with it which is set when a message arrives in a bounded `fifo` (i. e. one whose length is limited to a fixed number of messages) which is full, causing the oldest message in the `fifo` (the front message) to be lost. `Overflow()` returns the value of this flag, and causes the flag to be reset. Thus, it is similar to the overrun bit in a UART.

#### 4. Examples

To clarify the ideas presented so far, this section presents several examples of the use of the functions defined above. The first is a simple use of export and import fifo's to transport an array from one task to another. The second uses the size() and waitf() functions to implement a "server" task. Finally, the third example demonstrates the creation of multiple tasks from a single program to compute the dot product of two vectors.

##### 4.1. Exporting an Array

This example demonstrates how one can export an array of length N from one task to another. Two tasks, T1 and T2, respectively send and receive the array A. The sending task, T1, is defined as:

```
#define N 100          /* size of array */

T1()                  /* task to send array */
{
    int A[N];          /* array to be sent */

    export ("A");      /* create export fifo A */
    getarray (A);      /* routine to input array A */
    put ("A", A, N*sizeof(int)); /* send array */
}

getarray(A)           /* routine to input array */
int A[];

{
    int i;
    for (i=0; i<N; i++) { /* input array */
        printf ("enter data:");
        scanf ("%d", A + i);
    }
}
```

The receiving task, T2, is defined as:

```
T2()                  /* task to receive array */
{
    int B[N];          /* array to hold result */

    import ("A", 0);   /* create import fifo A */
    get ("A", B);      /* get array */
}
```

The bootstrap routine for this pair of tasks is defined as:

```
#include <stdio.h>

bootstrap()

{
  int T1(), T2();          /* names of the tasks */

  mktask ("T1", T1, 1, NULL, 0);
  mktask ("T2", T2, 2, NULL, 0);
}
```

#### 4.2. A Server to Square Variables

A "server" is implemented which will square and return any floating point value sent to it. It is assumed that each task which uses the server has an export and an import fifo whose name is "A[i]", where i is some integer assigned to that task. To use the server, a task sends a floating point number on its export fifo, and waits for the result to arrive on its import fifo.

In this example, the server uses the waitf() routine to wait for data to arrive in one of N fifo's. It then uses size() to poll the fifo's to determine which one received the data. One could have used qlength() instead of size(), however size() is executed more efficiently by the simulator. Thus, size() is recommended when testing to see if a fifo is empty.

The server first sets up an array of character strings (actually, an array of pointers to character strings) which lists the set of fifo's it will waitf() on. This is followed by an infinite loop which consists of two actions:

- (1) wait for data to arrive in one of its fifo's
- (2) poll its fifo's, and return a result for any data that has arrived.

Thus, the server task SQUARE is defined as:

```
#include <stdio.h>
#define N 10

SQUARE()
{
  int i;
  char *p[N];          /* pointers to fifo names */
  char *cnvarrsv(), *cnvarr();
  float x;
```

```
exparr ("A", N);      /* create arrays of fifo's */
imparr ("A", N, 0);

/* set up array of fifo names for waitf() procedure */
for (i=0; i<N; i++)
    p[i] = cnvarrsv ("A", i); /* fifo name */

/* At this point, p[0] points to the character string
   "A[0]", p[1] to "A[1]", etc. */

for ( ; ; ) {        /* loop forever */
    waitf (p, N);      /* wait for data */

    /* Execute the following when data arrives.
       Find fifo with data and return result. */

    for (i=0 ; i<N ; i++)
        if (size (cnvarr("A", i)) != 0) {
            get (cnvarr("A", i), &x);
            x *= x;
            put (cnvarr("A", i), &x,
                sizeof(float));
        }
    } /* Infinite loop */
}
```

Note that cnvarrsv() is used instead of cnvarr() whenever the value returned is assigned to a variable. A task using the server might be defined as (in file USER.c):

```
USER(x)
char x[];

{
    float z;

    export (x);
    import (x, 0);
    .
    .
    .
    put (x, &z, sizeof(float));
    get (x, &z);
    .
    .
    .
}
```

The bootstrap routine must pass a parameter to the user task specifying the name of the fifo it is to use. Suppose we want to create N tasks from the USER program defined above. To do this, bootstrap() might look like:



```
bootstrap()
{
int i, id, strlen();
char *cnvrr();

id = 1;      /* id of created tasks */

for (i=0; i<N; i++)
    mktask ("USER", USER, id++, cnvrr ("A", i),
            strlen(cnvrr ("A", i)) + 1);

    mktask ("SQUARE", SQUARE, id++, NULL, 0);
}
```

The above code uses the predefined C routine `strlen()` to get the length of the string `s` (in bytes). A one is added to `strlen` because of the C convention of appending a null (`\0`) character to mark the end of character strings.

Finally, note that even though many tasks are created from the `USER` program, the `init()` routine need not be used (`SQUARE` does not use `init()` because multiple instances are not created). This is because `USER` does not have any global or static variables. The next example will demonstrate the use of `init()`.

#### 4.2.1. Dot Product

This example computes the dot product of a pair of vectors, `A` and `B`. Three programs are used. The first (called `T`) multiplies two array elements together. For arrays of length `N`, there are `N` tasks generated from this program. `T` uses global variables so that use of the `init()` routine can be demonstrated. The second, called `TDISTR`, distributes the data to these `N` tasks, and the third, `TRESULT`, collects the results and adds them up to form the dot product in the variable `sum`. One task is created for each of `TDISTR` and `TRESULT`.

The parameter and global data structures are defined first:

```
struct parm          /* format of parameters */
{
char x[10];          /* input file name */
char y[10];
};

struct               /* globals for task */
{
float a, b, c;       /* work variables */
}
```

```
{ glob;
```

Task T is defined as follows:

```
T(p)          /* multiply numbers */  
struct parm *p;      /* x and y are parameters  
                      specifying import fifo's */
```

```
{  
  init (&glob, sizeof (glob));  
  import (p->x, 0);      /* create fifo's */  
  import (p->y, 0);  
  export ("C");          /* result fifo */  
  
  get(p->x, &glob.a);     /* get operands */  
  get(p->y, &glob.b);  
  glob.c = glob.a * glob.b;  
  put ("C", &glob.c, sizeof (float)); /* result */  
}
```

```
#define N      100
```

```
TDISTR ()  
{  
  float A[N], B[N];      /* vector operands */  
  int i;  
  exparr ("A", N);        /* arrays of fifo's */  
  exparr ("B", N);
```

```
  .  
  .  
  .  
  <<input data into A[] and B[]>>  
  .  
  .  
  .
```

```
  for (i=0; i<N; i++) { /* loop to send data */  
    put (cnvrr ("A", i), A+i, sizeof(float));  
    put (cnvrr ("B", i), B+i, sizeof(float));  
  }  
}
```

Note that the put() routine takes a pointer to the data as an argument, and not the data itself. Thus, "A+i" and "B+i" are used rather than "A[i]" and "B[i]".

And finally, the task which collects the multiplied operands and generates the result, TRESULT, is:

```
TRESULT ()  
{  
  float sum, temp;  
  int i;
```

```
import ("C", 0);      /* import fifo's */

sum = 0.0;              /* add results from T */
for (i=0; i<N; i++) {
    get ("C", &temp);
    sum += temp;
}
}
```

Note that all the values of  $A[i] * B[i]$  collect in fifo "C" of TRESULT. The order in which the data arrives is arbitrary, but this is of no consequence here.

The bootstrap() routine for dot product would look like:

```
#include <stdio.h>

bootstrap()

{
    int i, id;
    int T(), TDISTR(), TRESULT();
    char *cnvrr();
    struct parm p;

    id = 1;

    /* create tasks from program T */

    for (i=0; i<N; i++) {
        strcpy (p.x, cnvrr ("A", i));
        strcpy (p.y, cnvrr ("B", i));
        mktask (cnvrr ("T", i), T, id++,
                &p, sizeof (p));
    }

    /* create other tasks */

    mktask ("TDISTR", TDISTR, id++, NULL, 0);
    mktask ("TRRESULT", TRESULT, id++, NULL, 0);
}
```

Note that the parameters passed to the tasks in the calls to mktask() do not contain pointers. Storage for the strings x and y is allocated within the parm structure itself, rather than using pointers to character strings. The predefined routine strcpy() is used to copy characters into the parameter list. Finally, notice the use of cnvrr() in mktask() to create different names for the tasks defined using the program T.

## 5. TIMING MODEL

A timing model is clearly necessary if the simulator is to be used for any kind of performance evaluation, whether of communications networks or execution of parallel algorithms. This section will describe the timing model provided by the simulator as well as a high level view of how it works.

Each processor has a clock which keeps track of time for the task running on that processor (recall that each processor is assumed to execute at most one task). In the real multiprocessor, all of the clocks would advance simultaneously with real time, and thus would (at least ideally) indicate the same time. Here however, execution of tasks is time multiplexed on the host computer running the simulator. Thus, at any moment in the simulation run, clocks on different tasks will usually differ. The clock for each task will reflect how far that task has progressed in its computations.

When the simulator begins executing a task, the clock for that task is started. At some later time, when execution is switched to another task, the original task's clock is stopped. A task's clock also advances if the task becomes idle, e. g. when the task must wait for data to arrive from some other processor. All tasks initially begin execution with their clocks set at "0".

Using this clock, the simulator can estimate when events occur in the real system, such as when tasks send messages, when messages arrive (actually, the switch model determines this), etc. The simulator also uses the timing model to accumulate statistics on the task, such as how long it spent executing, and how long it was waiting for data. Such statistics are printed out at the end of each simulation run.

Implementation of the timing model requires that the user compile his programs with a modified version of the C compiler (cc). Let us call this modified version simcc. Simcc is like cc except that in addition to generating VAX assembler instructions for the code being compiled (which are automatically piped into the assembler program), it inserts additional instructions which will increment the clock for the task as it executes. Thus, at least conceptually,

execution of each assembler instruction generated by cc is followed by the execution of an inserted instruction which causes the clock for that task to advance by an amount of time equal to the execution time of that instruction. An option to simcc allows the user to assign an execution time to each VAX instruction (i. e. each opcode). Use of this option is described in Appendix IV. The current implementation assigns a default execution time of one microsecond to each instruction if no such assignment is made.

At present, there is one routine available to the user pertaining to the timing model. This is the getclk() routine, which allows a task to read it's own clock. Getclk() takes a single parameter, the id of the executing task. It returns an integer which is the number of microseconds that have elapsed since the task was created (recall that all tasks are assumed to be created at time 0).

## **6. THINGS TO LOOK OUT FOR**

Unfortunately, the simulator could not be designed in a manner that would detect all possible errors the user might commit. Because of this, some rather subtle bugs may arise. This section outlines some "easily made" mistakes the user might encounter in writing programs for the simulator.

### **6.1. Global and Static Variables**

As pointed out earlier, if multiple instances of a task are created, and if the program for these tasks uses global or static variables, the `init()` function **MUST** be used. Since the simulator cannot determine which tasks fall under this category, it cannot detect those which neglect this detail.

Tasks neglecting to use `init()` when they should may find that variables mysteriously change without being assigned to! This is because the simulator will not save or restore the values of these variables when execution switches from one task to another. Thus, in effect, all of the tasks created from that program use the same sets of variables, allowing one task to change another's. Needless to say, the results are generally unpredictable.

### **6.2. Use of `Cnvarr()`**

The user should never assign the value returned by `cnvarr()` to a variable. Subsequent calls to `cnvarr()` will change the value of this variable, again without the use of an assignment statement. As noted earlier, this is because the result generated by `cnvarr()` is always stored in a single buffer (actually, each task has its own buffer). Thus, each call overwrites the value generated by the previous call.

If the value is to be assigned to a variable, `cnvarrsv()` should be used. The result is then stored in a data area created via the `calloc()` function. It is left to the user to release this storage (via `cfree()`) when he is done using it.

### **6.3. Variable Sharing**

Two tasks should never access the same variables. Doing so effectively allows tasks to communicate without using the switch model, thus invalidating any statistics gathered during the simulation run.

### **6.4. Fifo Names**

Blanks imbedded within fifo names are significant. Thus, "T ", " T", and "T' are three distinct fifo's. Thus, a message placed in export fifo "T" will not be sent to import fifo's " T" or "T'".



## 7. IMPLEMENTATION

This section describes the implementation of Simon. It is intended for a programmer who is planning to modify or augment Simon, or for someone interested in developing new switch models. A high level description of the operation of Simon is described, as well as details of the manner in which the program is physically partitioned into modules (i.e. files) and the interfaces and interactions between modules. Readers interested in low level details (e.g. the order and types of parameters in subroutine calls) should refer directly to the source code for Simon. It is assumed that the reader is familiar with the user's interface to Simon (seen by the applications programmer) described earlier. A general knowledge of the internal workings of compilers, operating systems, and event driven simulation programs would also be helpful, though not strictly necessary.

At the highest level of abstraction, Simon models the multicomputer system as a sequence of *state transitions*. In Simon, internal variables keep track of the current state of the real system at any instant in time. Transitions are modeled as *events*. Just as the real system moves from state to state following certain well-defined state transitions, Simon's internal variables move from value to value following certain well-defined events. The model used by Simon to represent the state of the multicomputer system, and the events modeling transitions between states are defined here.

Simon is implemented as a discrete time, event driven simulation program. It is *discrete time* (in contrast to continuous time) because the set of times at which state transitions, i.e. events, can occur is a countable (and finite because the precision of the host computer is limited) set. It is *event driven* because Simon executes by processing events, and stops when there are no more events to process. Each event is timestamped to indicate the time at which the event occurs in the real system. Events are processed in time increasing order, and generally causes some modification in the state of the system and generate, or *schedule*, additional events. Thus the simulator always has a backlog of events waiting to be processed. It continually tries to clear this backlog by processing

events, one after another, however in doing so, keeps adding to the backlog! When the backlog is finally cleared, the simulation run is complete.

The operation of Simon consists of setting up some initial state, and then processing events to model the state transitions of the real system. Roughly speaking, Simon is composed of the following components:

- (1) Events.
- (2) Tasks.
- (3) Fifos and Virtual Circuits.
- (4) Messages.
- (5) Timing Model.

A high level understanding of the internal workings of Simon can be obtained by understanding the events defined by Simon, and how they modify the state of the system. These events will be discussed first. The remaining components model the state of the multicomputer system, and will be discussed in subsequent sections.

First let us say something about the overall organization of the Simon program. The source code is distributed across several files, with each file containing the code for performing some logical function. The code for managing tasks for example, reside in the file "task.c". The different files and the functions they perform are listed in Appendix II. Interactions between files is, with only a few exceptions, through procedure calls. There are very few global variables. Although this results in some performance penalty since procedure calls are more expensive than accesses to global variables, it improves the modularity and readability of the resulting code. The emphasis in developing Simon was towards modularity to promote readability and to reduce the difficulty of making changes to the code. Modifications to improve performance have not been attempted at the time this document was prepared, but is certainly an area of future work.

Throughout this document, the convention will be used that when the name of a routine is referred to, the file in which it resides will follow in bold-face

print. For example, `save()` (`task.c`) refers to the `save` routine, which resides in the file `task.c`. Routines residing in the switch model will be assumed to reside in a file called `swmod.c` (the simplest switch model), unless specified otherwise.

### 7.1. Events

State transitions in the real system are modeled in Simon by events. A data structure called the *event queue* keeps a time sorted list of events waiting to be processed. The `main()` (`mainsim.c`) program of Simon repeatedly executes the following actions:

- (1) Take the next event off of the event queue.
- (2) Processes the event by modifying some of Simon's internal variables and/or scheduling other events.

Execution terminates when the event queue becomes empty.

The real multicomputer system consists of a number of autonomous computers, each executing some part of a parallel application program. How does one go about selecting the events for Simon? Removing an event from the event queue, processing it, and then scheduling other events is a fairly time consuming process, so it is unreasonable to go through this scenario each time the application program modifies one of it's variables. Instead, Simon allows each task to execute, modifying it's own variables. Each task executes directly on the host computer. Thus, under this scheme, only one type of event is required: an event which initiates execution of a task. When the simulation begins, the event queue is initialized to hold one such event for each task defined by the application program, and each task is executed to completion, one after the other. In Simon, such an event which marks the initialization of a task is called an "init-task event", and as just described, one is scheduled for each task created by the "bootstrap" program as part of the initialization process.

After a little thought however, it is easy to see that this simple scheme will not work if tasks interact with each other, e.g. by exchanging messages. Clearly a task cannot execute to completion if uses data generated by another task which has not yet begun execution. Thus, other events are necessary to

simulate interactions, and to force one task to temporarily stop executing while other tasks are allowed to "catch up". Since all interactions between processors occur via messages, let us define two new events:

- (1) The "send event" denotes a processor sending a message.
- (2) The "arrive event" denotes a message arriving at a processor.

The send event is generated when a processor executes the `put()` (`lib.c`) routine. Since a send event denotes a message entering the communication switch, it must be processed by the switch model. As a consequence of processing the send event, the switch model eventually schedules an arrive event indicating that the message has reached its final destination.

Getting back to our original scheme, we see that the two additional events described above are still not sufficient, since we still need a mechanism to temporarily stop execution of tasks on the host computer so that others can execute and generate the data they need. One more event, called the "get event", is defined for this purpose. When a task needs data from another task, it calls a routine defined by Simon, e.g. the `get()` (`lib.c`) routine. Simon temporarily stops execution of the task, and schedules a get event to note the fact that this task has been stopped. Simon then goes back to the event queue and processes other events. Eventually, when data arrives for this task, it will be restarted, and allowed to continue execution.

Consider the sequence of events that occur when two tasks are created, A and B. Suppose task A sends a single message to task B. Because of timing variations, the scenario which will now be described is not the only one possible, however it illustrates the interactions of the events described above. Initially, the event queue consists of two events, namely the `inittask` events for A and B. Suppose that the `inittask` event for task B appears ahead of that for A in the event queue. Task B begins execution, and requests to receive the message by executing the `get()` (`lib.c`) routine. Simon schedules a get event, and blocks the task. The next event is the `inittask` event for A, so A begins execution. Task A sends the message by executing the `put()` (`lib.c`) routine. This causes a send

event to be scheduled into the event queue. Task A continues to execute, since as will be seen later, there is no reason to stop it now. Let us assume that A completes execution, returning control back to Simon. Returning to the event queue, we see there are two events: the get event denoting B's request to receive the message, and the send event denoting the message being sent. Let us assume the send event precedes the get event in the queue. Since this event signals a message entering the switch, the switch model is called upon to simulate the transmission of the message. In doing so, the switch model can perform arbitrarily complex (or arbitrarily simple) operations, perhaps scheduling some of its own events. The final result is that it schedules an arrive event which indicates that the message has reached its destination. Now the queue holds an arrive event and B's get event. The two possible ordering of these events in the queue correspond to the two situations which can occur in the real multicomputer: either the message arrives before B asks for it, implying B does *not* have to wait for it, or the message arrives after B asks for it, and B must become idle, waiting for data.

Consider the first case. Here, the arrive event is ahead of the get event. Processing of this event merely corresponds to Simon noting that the message has arrived. It places the message in the appropriate import fifo. The get event is then processed. The message is removed from the fifo and passed to task B. The task is restarted, and completes execution. Since the event queue is now empty, the simulation is complete.

Consider the second case. Here, the get event precedes the arrive event, indicating the task asked for the data before it had arrived. In the real multicomputer, this corresponds to the task blocking, waiting for data to arrive. Here, Simon marks the task as "blocked", and goes back to process the next event in the event queue. The only remaining event is the arrive event. Simon notices that task B is blocked, waiting for the arrive event, so it unblocks the task, passes it the message, and allows it to continue execution. Again, B completes execution, and the simulation is complete. It is important to distinguish between the two types of "blocking" performed by Simon. In the first, the task

was blocked to allow another task to "catch up". This "blocking" is just an artifact of the simulation technique, and does not correspond to a task being blocked in the real system. The second blocking corresponds to a task waiting for data in the real system. Note that in Simon, the first type of blocking is marked by a get event in the event queue, while the second type of block does not have any such events scheduled. The task is later restarted by an arrive event.

With the scheme described above, each task has its own clock which indicates how much time has elapsed since the task began execution. It is important to realize that at any given time in the simulation, different tasks will usually have different values on their respective clocks. This points out another important function of the event queue: namely to ensure that interactions among tasks are simulated in the proper time sequence. Since some task's clocks are ahead of others, it is important not to let a task get "too far" ahead. For example, suppose tasks "A" and "B" each send a message to task "C" using some fifo, say "foo". Clearly, in the real system, either the message from A will arrive at C first, or the message from B will arrive first. Suppose the one from A arrives first. Suppose B was allowed to execute, scheduling a send, and eventually an arrive event for it's message. Now suppose C executed. It executes a get() (lib.c), and erroneously receives the message from B. Only later is it discovered that A generated a message which arrives before the message from B, and that the simulation has been compromised! Happily, the scenario described above cannot occur in Simon. The simulation cannot be compromised if it executes interactions between tasks in the proper time sequence. However, since all interactions go through the time sorted event queue, events residing in the queue cannot be simulated in the wrong order. The only way events can be simulated out of sequence is if an event is scheduled with a timestamp preceding that of the event now being processed. However, this implies an event causes another event which occurred earlier in time than the first, e.g. a message arriving before it was sent! Since this cannot happen in the real system, it cannot happen in the simulator, so long as events correspond to occurrences in the



real system.

From this perspective, the purpose of the event queue takes on a new light. The event queue is no longer simply a "warehouse" of events waiting to be processed. In addition, it acts as a sequencer which ensures that events modeling the real system are processed in the correct time sequence. Thus, it is easy to see when Simon must temporarily block a task, and when it may allow the task to continue executing: when a task's behavior may be affected by an interaction with another task, Simon must block the task and schedule an event on the event queue. Otherwise, the task may be allowed to continue executing. Using this rule, it is now clear that we do not need to block a task executing a put() (lib.c), and similarly, we do not need to block a task executing a get if the corresponding import fifo holds at least one message. The latter results from the fact that Simon places data into each import fifo in correct time sequence (since only arrive events cause additions to import fifo's, and all arrive events are guaranteed to be sequenced correctly), so no new message will be placed in front of another message already in the fifo.

Finally, one other event is defined. This is the "initsw" event. This event occurs exactly only once, at the beginning of every simulation run, and is used to initialize the switch model.

In summary, five different types of events have been defined:

- (1) inittask: initiate execution of a task.
- (2) send: send message into communication network.
- (3) arrive: message arrives from the communication network.
- (4) get: task pauses because it interacts with another task.
- (5) initsw: initialize switch model.

Each of these will now be described in greater detail. When these events are removed from the event queue, the main program calls a particular routine to process that event. The name of this routine and the functions it performs will be discussed.



#### 7.1.1. The Inittask Event

Simon initially places an inittask event in the event queue for each task created in the bootstrap() routine. This event is processed by the routine init-task-ev() (evhand.c). It calls the routine sttask() (mainsim.c) which sets some flags indicating that the task is to begin execution. Control is then returned to the main program. Note that sttask() does *not* begin execution of the task directly, but only sets some flags representing a request to start the task. After processing each event, the main program checks to see if the event it has just finished processing requested to start a task. If so, it begins executing the task via an ordinary procedure call. Restarting tasks (after being stopped because of interactions with other tasks) also follow this same procedure, i.e. the event handler sets flags to request the restart, and the task is later restarted by the main program. By using this indirect scheme for starting/restarting the execution of tasks, *all* tasks begin execution from the same point in the main program. This ensures that the base of the runtime stack always occupies the same memory location, and it simplifies the task switching operation because all returns from the task (either to stop it temporarily or permanently) resume execution at the same place. After a task stops execution, the next function performed is the removal and processing of the next event in the event queue.

#### 7.1.2. The Send Event

The send event is scheduled by the put() and puts() (lib.c) routines, and is processed by the sendev() (swmod.c) routine. The functions performed by this routine will be described later in the discussion on switch models.

#### 7.1.3. The Arrive Event

The arrive event is scheduled directly, or indirectly through the sendev() (swmod.c) routine, and is processed by arriveev() (evhand.c). The event handler must add the message to the destination import fifo. Also, if a task is blocked, waiting for this message to arrive, the sttask() (task.c) routine is called to request that the task be restarted.

#### 7.1.4. The Get Event

A get event may be scheduled by any routine which has an outcome depending on some interaction with another task. Currently, the routines which fall into this category are: `get()`, `overfl()`, `qlength()`, `size()`, and `waitf()` (`lib.c`). These routines may or may not cause a get event to be scheduled. If it can be determined that the result of this routine is not affected by anything any other task can do, then a get event need not be scheduled. For example, if the `get()` routine is executed on a `fifo` which already holds at least one message, then this routine will always return the first message in the `fifo`, regardless of any other messages that may arrive. The `get()`, `size()`, and `waitf()` routines do not schedule a get event if the import `fifo` specified by the routine (or in the case of `waitf()`, at least one `fifo`) has a message. Otherwise, a get event is scheduled. The `overfl()` and `qlength()` routines always schedule a get event because they must wait until all other tasks have "caught up" with the current one in order to determine the result returned by the routine.

Let us now consider the processing of the get event when it appears at the head of the event queue. Five different types of get events exist, distinguished by the routine which caused the event. All types are processed by the `getev()` (`evhand.c`) routine. If the event was scheduled by the `get()` routine, the task is restarted if the `fifo` the get was performed on now has a message. The restart request is performed by calling `sttask()` (`task.c`), as described above. In this case, the message arrived before the task needed it. Otherwise, the task is marked "blocked" via the `blktask()` (`task.c`) routine signifying that the task is idle, waiting for data in the real system. The "waitf" get event is handled in a similar fashion. The remaining get event types all cause the task to be restarted, since they only ask for information which can now be determined with complete certainty (e.g. the number of messages in a `fifo`).

#### 7.1.5. Other Event Related Routines

A number of other routines are defined relating to the event queue. These routines are defined in the file `evnt.c`. Each type of event has a separate routine

for scheduling an event, e.g. the `scsend()` routine schedules a send event. All call the `schedule()` routine which inserts the event into the time ordered event queue. Routines are provided to remove events from the event queue, and to return parameters for the different event types. The `initevq()` routine initializes the event queue, `notevqempty()` checks to see if the queue is empty, and the `prevq()` routine prints the contents of the event queue onto the standard output. This last routine is provided for debugging purposes. Finally, a number of other routines are provided for internal storage management purposes.

## 7.2. Tasks

Simon maintains information on each task to allow them to time multiplex their execution on the host computer, and to collect statistics. In many respects, the management of tasks on Simon resembles the management of processes in a uniprocessor operating system. First, Simon must be able to create and begin the execution of a task when an `inittask` event occurs. When a `get` event occurs, the task must stop execution, and another task must be restarted. Finally, when the task completes execution, it must be destroyed.

### 7.2.1. Starting Tasks

Creating and beginning the execution of a task is a relatively straightforward operation. When the user-defined `bootstrap()` routine creates a task by executing the `mktask()` (`lib.c`) routine, Simon creates and initializes a data structure for the task called a "task control block", or `tcb`. The information maintained in the `tcb` includes the name of the task (a user defined character string), the status of the task (e.g. running, or blocked and waiting for data - if the task is blocked, then this means it is blocked in the real system), pointers to saved state information, a pointer to the code for the task, and various statistics on the execution of the task.

The `mktask()` (`lib.c`) routine specifies, among other things, an integer called the "id" of the task being created. A task's id is unique, and is used internally by Simon to refer to the task. Since tasks cannot be created dynamically, task id's are never reused. Once the `tcb` is set up, the task is "officially" created.

After `mktask()` creates the tcb by calling `taskcr()` (`task.c`), an `inittask` event is scheduled so that the task will eventually begin execution. When the `inittask` event is processed, the task begins execution through an ordinary procedure call. The `inittaskcv()` (`evhand.c`) handles the event. The mechanics of performing this function were described earlier in the section in `inittask` events.

### 7.2.2. Stopping / Restarting Tasks

Stopping and restarting the execution of a task requires some fancy "footwork" on the part of Simon. This is because a certain amount of information must be saved when the task stops executing, and restored when the task is restarted. The information which must be handled this way depends on the host computer, and the program used to compile application programs. Here, we will restrict our discussion to the C compiler (used at Berkeley) and the VAX host computer. In the discussion which follows, it is important that the reader distinguish between a *task* defined by Simon, and a *process* running under the host computer. Simon may contain several tasks, however to the host computer and operating system, it is run as a single user process.

A C program executing on the VAX uses the VAX's runtime stack to hold local variables, as well as information indicating the dynamic structure of procedure calls (so the correct procedure can be executed when the current one returns). A program executing as a single process uses a single runtime stack. Thus, the various tasks running under Simon all share the same runtime stack. When a task stops execution, the current runtime stack must be saved, since this information will be overwritten by the next task. Similarly, when a task is restarted, it's runtime stack must be restored before execution can be resumed.

In addition to the runtime stack, other information must be saved/restored if multiple instances of a task are created from the same program. Besides local variables, each task has a number of global variables. The C compiler assigns each global variable to a private memory location. As long as only one instance of a procedure is created, the memory locations assigned to the globals

remain unshared, so it is not necessary to save/restore them when tasks are stopped/restarted. If several tasks are created from the same program however, all will use the same memory locations for their global variables, so it is necessary to save/restore them. Unfortunately, Simon cannot easily determine where these variables are located. Thus, it is up to the user (via the `init()` (`lib.c`) routine) to tell Simon where its global variables are located if several tasks are created from the same program. It might be noted that dynamic variables created by e.g. the `calloc()` program need not be saved/restored, even if multiple instances of a task are created. This is because a private copy is created for each task after the task begins execution. Since these memory locations are not shared by other tasks, no saving/restoring is necessary.

Thus, stopping the execution of a task causes the task's runtime stack, and perhaps its global variables, to be saved in the task's tcb (actually a pointer to the saved area is kept in the tcb). This state must be restored before the task can be restarted. Let us consider the situation in which a task does a `get` on an empty fifo, and must temporarily stop executing. The `get()` (`lib.c`) schedules a `get` event, and then calls the `save()` (`task.c`) routine. `Save()` saves the runtime stack, and global variables if necessary. Now, we would like to return to the main program at the point where it starts/restarts tasks, so that we can go on to process the next event in the event queue. `Save()` cannot simply execute a return however, since this will take us back into the `get()` routine. In order to accomplish this, the return address information in the runtime stack is overwritten, so that instead of returning to `save`, we pop off all of the stack frames for the task, and return to the main program (recall that all tasks are started/restarted from the same point in the main program). The `resume()` (`task.c`) performs this covert deed. When it returns, the main program is now executing, rather than the program which called it, `save()`.

When it is time to resume execution of the task, the `restore()` (`task.c`) routine is called. All starting/restarting of tasks are performed by this routine. To restart the task which did the `get()`, the saved runtime stack is loaded back into the real runtime stack, and globals are restored if necessary. This is a

dangerous business, because in restoring the stack, the stack frame of the `restore()` routine is overwritten, destroying its local variables! Nevertheless, a return is now executed. Since the stack now reflects the state of the stack at the time at which `save()` was saving the stack, the return executes as if `save` did the return, i.e. we are now suddenly back in the `get()` routine at the point just after the call to `save`. From the viewpoint of the task executing the `get()` routine, it simply called `save()`, and sometime later execution returned from `save`. Thus, the `save` routine acts as a no-op to the program which executes it. Of course, in the time between the call to `save()` and its return, thousands of other events may have been processed.

From the description presented above, it is clear that the saving/restoring of a task is a tricky business using certain "unconventional" coding practices. Modifying the `save()`, `resume()`, and `restore()`, routines should only be done after a thorough understanding of its workings has been achieved. Seemingly harmless changes, e.g. altering the order in which variables are declared in the `resume()` routine, can have disastrous consequences.

### 7.2.3. Terminating Execution of Tasks

Terminating the execution of a task is straightforward. Excluding execution errors which terminate the entire simulation run, a task may finish execution by either returning from its main program, or by executing the `stop()` (`lib.c`) routine. Returning to the main program causes the runtime stack to be popped by normal procedure returns, so control is returned to Simon as a consequence of returning from the procedure call which initially caused the task to begin execution. Execution of the `stop()` (`lib.c`) routine causes the stack to be popped in a manner similar to that used in saving the stack, except the stack's contents are not saved since the task will not be restarted. In any case, execution resumes as if a return was executed from the initial call which began execution.

### 7.2.4. Other Task Management Routines

The remaining routines for task management (`task.c`) provide various book-keeping functions, e.g. keeping track of which tasks are blocked on which fifos,



or provide information about tasks, e.g. the name of a task given its id. Finally, the `prtask()` routine prints out summary information of the execution of all tasks. This routine is normally called only once, at the end of the simulation run.

### 7.3. Fifos and Virtual Circuits

The fifos represent the interface which carries messages between tasks and the switch model. Each import/export fifo pair of the same name corresponds to a virtual circuit. The `fifo.c` file contains routines to create import and export fifos, to add (remove) elements to (from) fifos, to test the status of fifos (full, empty, etc.), and to gather end-to-end traffic statistics. The implementation of the fifo management routines is described fully in [John81].

### 7.4. Messages

The file `task.c` creates the notion of tasks, and defines certain operations, e.g. saving state, which can be performed on them. Each task is identified with a unique id. Other modules need not know *how* the various functions are performed within the task module, but only need to be concerned with the interface it provides. Similarly, `msg.c` is a module defining how messages are dealt with.

Each message is identified by a unique id, called the message id. Unlike task ids however, a message id remains unique only as long as the message remains in existence. Once the message arrives at its final destination, the message id may be reused. This is necessary for efficiency reasons, since unlike tasks, messages come and go at a relatively high rate.

When one processor sends a message to another, a message must be created, and placed into an export fifo (actually only the message id is placed into the buffer since this is the interface provided by the message module). The data portion of the message must be copied into a separate buffer. A simple pointer to the task's variable cannot be used because the task might change this variable before the message arrives at its destination, causing the new value to be transmitted rather than the value which existed when the message was sent.



When the message reaches its destination, the message id is placed into an import fifo. When the receiving task reads the message (via the `get()` (`lib.c`) routine), the contents of the message is copied into one of receiver's variables, and the message is destroyed. The id is now free to be used by newly created messages.

The simple scheme described above is complicated somewhat by the fact that Simon allows broadcast, i.e. multiple destination communications. If a message is broadcast to several destinations, it is wasteful to create a separate copy and message id for each destination, so only one copy is created. A count is associated with the number of copies existing in the real system. When the message is created, the count field of the created message is set to equal the number of destination processors. When a copy of the message arrives at its destination, the count is decremented. The message is destroyed and its id released when the count becomes zero. Note that this scheme allows different physical messages in the network to have the same message id, however this does not lead to any inconsistencies within Simon.

Each message currently in existence has a message descriptor associated with it. This descriptor is analogous to the tcb allocated to each task. Message descriptors include such information as the size of the message, location of its buffer, time at which it was created, number of copies, etc.

Messages are created and their contents copied into a buffer through the `mkmsg()` (`msg.c`) routine. `Rmmsg()` (`msg.c`) destroys messages, and `cpmsg()` (`msg.c`) copies the contents of the buffer for the message into the receiver's data area. Other routines are provided to extract information about a particular message, e.g. its size.

### 7.5. Timing Model

The timing model consists of two components. The first is the `simcc` compiler which inserts instructions into the application program to continuously update a clock as the program executes. The second is the portion of Simon responsible for maintaining the current clock of each task. The implementation

of these two components will now be discussed.

#### 7.5.1. Simcc

First, the simcc program is the modified version of the cc program, the front end for the C compiler. The cc program calls the compiler to compile the program. The output generated by the C compiler is an assembly language version of the program being compiled. The assembler then creates an object file. The simcc program inserts a filter between the C compiler and the assembler. This filter is called the ccsf (C Compiler Statistics Filter) program. The output generated by ccsf is the assembly language program input to it with instructions which increment a global variable called "clock\_". Conceptually, an add instruction could be inserted before each assembly language instruction to increment clock\_ by an amount equal to the time to execute the instruction which follows. With a little thought however, it is easy to see that this scheme can be improved upon: a block of assignment statements in which the only entry is into the first statement, and the only exit is from the last statement may be preceded by an add instruction which increments the clock by an amount equal to the execution time of the block of statements. This latter scheme reduces the amount of overhead required to increment clock\_.

This is the approach taken by ccsf. The ccsf program also allows the user to specify the execution time of each VAX instruction, and computes the execution time of a block of instructions by simply computing the arithmetic sum of the execution times of the individual instructions. It turns out however, that one cannot arbitrarily insert add instructions into the instruction stream, since doing so may cause the condition codes on the host computer to change. The ccsf program, which is VAX specific, inserts instructions which first save the condition codes on the runtime stack, then inserts an add instruction, and finally inserts instructions which restore the condition codes. It is interesting to note that the final instruction which restores the condition codes is the "return from interrupt" instruction.

### 7.5.2. Timing Model Within Simon

The interface between Simon and the task's timing model (as set up by `simcc`) is the variable `clock_`. When a task first begins execution, `clock_` is set to 0. As the task executes, `clock_` is incremented via the inserted instructions. When the task stops executing (e.g. because it must wait on a `get`), the value of `clock_` is save in the task's `tcb`. `clock_` is restored to this value when the task is restarted.

Two timing models exist. In order to avoid anomolous effects caused by floating point arithmetic, `clock_` is implemented as an unsigned integer (32 bits on the VAX) variable. This means it can grow as large as approximately 4 billion time units. If the time unit is nanoseconds, this allows each task's clock to run up to approximately 4 seconds. With a microsecond time unit, clocks may reach values exceeding 4,000 seconds. The nanosecond timing model is in the file `timen.c`, while the microsecond model is in `timeu.c`. Only one of these two files is used on each simulation run.

Except for the switch model, the interpretation of time units is arbitrary, i.e. since the only real time in Simon is that expired by the tasks, no inconsistencies result. In the switch model however, delays refer to some time unit, so this module must know whether the time unit it is to compute for (say) delay is in microseconds or nanoseconds. To handle this situation, each timing module provides a number of conversion routines so to convert microseconds/nanoseconds to whatever time unit the rest of the simulator is using. This gives the switch models the flexibility to use whatever time unit they prefer, while remaining consistent with the rest of the simulator in a transparent fashion.

In addition to conversion routines, the timing modules provide a number of other routines for initializing, setting, starting, stopping, and reading a task's clock. Care must be taken in manipulating these clocks to ensure that time does not flow backwards! Should this occur, the simulator will be compromised, and unpredictable behavior results.

### 7.6. Switch Model Interface

Now that the events have been defined, the interface provided to the switch model can be explained. From the discussion above, it is clear that only three event — the send, arrive and init`sw` events — pertain to the switch model. Minimally, the switch model must provide three routines:

- (1) `initsw()` (`swmod.c`).
- (2) `sendev()` (`swmod.c`).
- (3) `switchev()` (`swmod.c`).

The `initsw()` (`swmod.c`) routine is called once at the beginning of the simulation run. It indicates such parameters as the maximum number of tasks that will ever be created, maximum number of import and export fifos, etc. It is responsible for initializing any data structures used by the switch model. Also, it is here that the switch can input any information about the communication network, e.g. the speed of communication links, topology of the network, etc. An input file pointer is passed to the routine. This pointer points to a file descriptor for the file specified in the command line which invoked Simon if the `-s` option was used. Otherwise, the standard input is used. The `-s` flag should *always* be used to input switch model information, since application programs often use the standard input to input their own data.

The `sendev()` (`swmod.c`) routine is called to process a send event which was scheduled by a task executing either the `put()` (`lib.c`) or `puts()` (`lib.c`) routines. The routine is passed information indicating the time at which the message was sent, the task sending the message, a list of tasks which are to receive the message, a message identifier (to be discussed later), and the size of the message in bytes. Note that a particular message may have several destinations since broadcast communications are allowed. Since sending a message corresponds to adding an entry to an export fifo, `sendev()` (`swmod.c`) must remove the message from the fifo, and schedule an arrive event for each destination. Each arrive event is timestamped to indicate the time at which the message arrived at its destination. The difference between this time the time at which the send

event occurred indicates the time required by the interconnection switch to transmit the message.

It might be noted that the `sendev()` (`swmod.c`) routine need not directly perform the functions described above. It is the switch model as a whole which is responsible for doing this. The `sendev()` (`swmod.c`) could schedule its own internally defined events which lead to removing the message from the export `ffo` and scheduling the associated arrive event(s). In a store-and-forward network for example, intermediate events might indicate messages going from hop to hop through the network.

The switch model is allowed to schedule and process its own, internally defined events. When such an event appears at the head of the event queue, the `switchev()` (`swmod.c`) routine is called. In general, the main program calls `switchev()` (`swmod.c`) whenever it removes an event from the event queue which it does not recognize (the five events listed earlier are the only events recognized by the main program).

Finally, one other restriction is placed on the switch model: messages sent on the same virtual circuit from the same task *must* be scheduled to arrive in the same order in which they were sent. This is because the user interface specifies that sequentiality is preserved on each virtual circuit. Violation of this rule will result in incorrect computations for application programs which depend on this feature.

#### 7.7. Notes on Porting Simon: Machine Dependencies

Although work is under way to port Simon to other machines, at present, no working version is known to exist on any other machine than a VAX. There are (at least) two portions of Simon which must be modified if it is to be ported to another machine. These are the `simcc` compiler, and the task swapping mechanisms.

Since the `ccsf` program scans through assembly language programs and assigns execution times to blocks of VAX instructions, it is clearly VAX specific. A new `ccsf` program is required for each host computer Simon is run on. The

function performed by ccsf remains the same, however different instruction opcodes will have to be recognized, and different instruction sequences will have to be inserted to ensure that the instructions can be inserted without disturbing the original code. All of this assumes that the compiler on the new host generates an assembly language program as an intermediate step. Without this, incorporating a timing model which can be used easily by Simon is much more difficult.

As discussed earlier, the code in task.c for saving and restoring the runtime stack of application programs depends on the format of the stack frame. This depends on the compiler and host machine. If Simon is ported to another machine, these routines will have to be modified to use whatever format that machine uses.

Finally, a third point of difficulty may arise if an attempt is made to move Simon to a machine which does not support 32 bit integer arithmetic. This is the timing model. Without 32 bit arithmetic, the granularity of each time unit may become too coarse. For example, if 16 bit arithmetic is used, clocks can only reach a maximum value of 65,000. With the low-precision microsecond time unit, clocks may only reach values of 65 milliseconds. Coarser time units are required to reach higher values, reducing the precision of the simulator even further.

## REFERENCES

- [John81] L. Johnson, "A FIFO Based Communication Scheme for a Multiprocessor Simulator," Master's Report, UC Berkeley (Sept. 1981).
- [Kern78] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," . Prentice-Hall Inc., Englewood Cliffs, New Jersey (1978).



## APPENDIX I

This appendix outlines the routines defined within the simulator which are available for use by the programmer.

### **char \*cnvart (base, i)**

char \*base; int i;

Convert character string p to an array reference using base name "base" and index value "i".

### **char \*cnvartsv (base, i)**

char \*base; int i;

Same as cnvart() except storage for the result is allocated via ealloc().

### **exparr (base, n)**

char \*base; int n;

Create an array of "n" fifo's with base name "base".

### **export (name)**

char \*name;

Create a single export fifo called of name "name".

### **get (name, datap)**

char \*name, \*datap;

Get next message from fifo "name", and store its data where "datap" points.

### **int getclk (id)**

int id;

Returns time on clock (microseconds) for task whose id is "id".

### **imparr (base, n, maxl)**

char \*base; int n, maxl;

Create an array of "n" import fifo's with base name "base", each holding up to "maxl" messages (infinite if "maxl" is 0).

**import (name, maxl)**

char \*name; int maxl;

Create a single import fifo with name "name", capable of holding up to "maxl" messages (infinite if "maxl" is 0).

**init (glob, lngth)**

char \*glob; int lngth;

Use when multiple tasks are created from a single program using global or static variables which occupy "lngth" bytes starting at location "glob".

**mktask (name, cdptr, id, parm, lngth)**

char \*name, \*parm; int (\*cdptr)(), id, lngth;

Create a task called "name", and assign it to task id "id". The block of memory "lngth" bytes long starting at "parmp" is passed to the procedure "cdptr" when the task starts executing.

**int overfl (name)**

char \*name;

Return and reset overflow flag for import fifo "name".

**put (name, datap, lngth)**

char \*name, \*datap; int lngth;

Put "lngth" bytes of data starting at location "datap" into export fifo "name".

**puts (name, datap, lngth)**

char \*name, \*datap; int lngth;

Same as put(), but send message to self as well.

**int qlength (name)**

char \*name;

Return number of messages currently in import fifo "name".

**int size (name)**

char \*name;

Return size of first message in import fifo "name", or 0 if fifo is empty.

**stop()**

Terminate execution of task.

**int task()**

Returns the calling task's id.

**char \*taskname (id)**

int id;

Returns a pointer to the name of the task whose id is "id".

**waitf (names, n)**

char \*\*names; int n;

Wait for data to arrive in one of the "n" import fifo's specified by "names".

## APPENDIX II

This appendix describes the mechanics of using the simulator. All object files may be found in the directory (on ESVAX and CSVAX) ~fujimoto/S. Example user programs are in the directory ~fujimoto/SIMEX. All test programs should be compiled with simcc (~fujimoto/BIN/simcc) rather than cc.

The "S" directory contains a set of object files. To use the simulator, link these object files to your test programs. The simulator's object files and the functions they perform are listed below.

object file	function
evhand.o	event handler routines (called by mainsim)
evnt.o	event queue primitives
fifo.o	fifo management routines
lib.o	user callable routines
mainsim.o	main program (initialization and main loop)
msg.o	message management routines
task.o	task management (saving/restoring state, etc.)
time.o	task timing model
util.o	miscellaneous utility routines
swmod.o	null switch model (zero latency transmissions)
ether.o	ethernet switch model

Note that the last two object files are switch models. Only one of these should be loaded on each simulation run.

### APPENDIX III

This appendix describes the parameters which can be specified in the command line when invoking the simulator. The parameters the simulator will accept are listed below.

flag	type of parameter	default	meaning
-nt	integer	200	max nmbr of tasks
-nm	integer	2000	max nmbr of msgs at one time
-nn	integer	1000	max nmbr of different fifo names
-ni	integer	1000	max nmbr of import fifo's
-ne	integer	1000	max nmbr of export fifo's
-t	<none>	off	generate traffic statistics
-o	file	stdout	output file
-s	file	stdin	input file to switch model

For example, to increase the maximum number of tasks to 400 while diverting output to the file foo, enter:

```
%sim -o foo -nt 400
```

#### APPENDIX IV

This appendix describes how to use the modified version of the C compiler. The simulator's timing model requires that all user programs be compiled with this compiler, called `simcc` (`~fujimoto/BIN/simcc`). In addition to all of the options provided in `cc`, `simcc` provides a `-T` option for specifying the execution times of VAX assembler instructions. When specified, the `-T` is followed by the name of a file which lists these execution times in the format described below. Thus, if your execution times are in the file "time", you would say:

`%simcc -T time ...`

to compile your program, where "..." is other options and names of files being compiled. Note that the blank after `"-T"` must be present. If the `"-T"` option is not specified, the default execution times will be used (all instructions execute in one microsecond).

The file "time" consists of a list of pairs, with each element of the pair separated by blank(s) and/or tab(s), and successive pairs separated by blank(s), tab(s), and/or newline(s) (elements of a pair must be on the same line). The first element of a pair is the mnemonic for a VAX assembler instruction. The second element is an integer specifying the execution time of that instruction in NANOSECONDS. An example "time" file is:

```
pushl    200
calls    200
addl3    200
muld3    200
movd     200
```

which specifies that the five instructions above execute in 200 nanoseconds. Note that mnemonics should all be in small letters, and the execution time is an integer with NO decimal point. All instructions besides these five will be assigned the default execution time (which is still set at 1000, or 1 microsecond).

# SELF-CHECKING VLSI BUILDING BLOCKS FOR FAULT-TOLERANT MULTICOMPUTERS

Yuval Tamir and Carlo H. Séquin

Computer Science Division  
University of California, Berkeley, CA 94720

## Abstract

The use of self-checking nodes and links for implementing fault-tolerant VLSI multicomputers is proposed. The system is composed of a large number of VLSI computers interconnected by high-speed dedicated links. Hardware that performs error detection is combined with system-level protocols that handle error recovery and fault treatment.

The self-checking nodes notify the rest of the system when their output is erroneous. In order to achieve high fault coverage, error detection is accomplished by duplication and matching. The critical circuit in this scheme is a comparator which must not be susceptible to faults that can remain undetected and later mask the failure of the functional modules. With both NMOS and CMOS technologies it is possible to implement a *self-testing* comparator that will produce an error indication if it incurs any single physical defect.

## Introduction

High reliability and high performance are primary goals of most computer systems. There are fundamental limits to the increases in reliability and performance that are achievable by improvements in technology alone. The limits on performance can be overcome by exploiting parallelism while the limits on reliability can be overcome by using *fault tolerance* techniques.

Parallelism can be exploited by a system that consists of a large number of *computation nodes*, each able to execute a subtask of the problem being solved. A possible architecture for such a system, which is compatible with the constraints of VLSI, is to interconnect these computation nodes by high-speed dedicated links and *communication nodes* that provide hardware support for communication functions such as message routing. Each computation node is connected to one of the communication nodes. A communication node has several *ports* through which it is connected to computation nodes and other communication nodes. We call such a system a *multicomputer*. The nodes and links are components (building blocks) that can be used to construct multicomputers with a wide range of sizes and topologies.

System *failure* occurs when the multicomputer doesn't perform according to its specifications at its interface with the "outside world." System failure is often the result of a failure of one of its components. Fault tolerance techniques attempt to prevent component failure from leading to system failure. A multicomputer is especially well suited for reliability enhancement using fault tolerance techniques since it is partitioned into independent and "intelligent" components (the computation and communication

nodes). Fault-free components can adapt to changes in faulty components and continue their operation in a way that leads to correct system output despite the fault.

A brief overview of techniques for implementing fault tolerance in a multicomputer is presented and the considerations that lead to the choice of an approach based on self-checking components are discussed. *Duplication and matching* is shown to be an effective practical technique for implementing nodes that are self-checking with respect to any likely fault.

## Implementation of Highly Reliable Multicomputers

The reliability of any system can be enhanced by increasing the reliability of its components through *fault prevention*<sup>1</sup> techniques such as specialized design methodologies, stringent quality control, and extensive validation and testing. These techniques typically result in more complex designs, greater cost, and lower performance. Furthermore, the effectiveness of these techniques is limited by our inability to exhaustively test complex VLSI chips.<sup>2</sup>

Alternatively, the reliability of the components can be increased by employing *fault tolerance* techniques. These techniques attempt to ensure that each component will continue to perform according to its specifications despite faults. Unfortunately, no component can tolerate an unbounded number of faults. The contamination of the system by incorrect output from a faulty component can be prevented only if, at some stage, other system components find out about the failure of the component and physically or logically isolate it from the rest of the system.

At the system level, software (protocols) can be used to detect and recover from the failure of components. For example, identical tasks may be assigned to three nodes and a "majority vote" taken on the results. One of the problems with this approach is that if the results conflict, it may be very costly or impossible to locate the cause of the discrepancy. Additional problems are the high overhead in computation resources and communication bandwidth and difficulties in effectively handling transient faults.

If a node fails due to a transient fault, it should be reset to a "sane state" and remain active rather than be removed from the system. If neighboring nodes are responsible for detecting such a failure, they must be given the authority to initiate the reset. However, this authority also allows a *failed* node to reset operational neighbors. In order to prevent this situation, each node must be responsible for its own reset. Hence, the node should include a mechanism to detect its own *erroneous states*<sup>3</sup> and initiate the reset.

Some of the deficiencies with the aforementioned techniques can be overcome by implementing fault tolerance in a VLSI multicomputer using hardware *error*



detection in conjunction with system level protocols which perform error recovery and fault treatment.<sup>1</sup> Errors caused by faults in the communication links are detected through the use of error-detecting codes. All nodes are self-checking and signal to the rest of the system when their output is incorrect so that it will not be accepted as correct. In addition, failed nodes attempt to reset themselves and reestablish a sane state. The immediate neighbors are informed whenever a node fails. If the node doesn't reset itself or fails too often, the neighbors can logically remove it from the system by refusing to communicate with it. The diagnostic status information is distributed throughout the system so that, eventually, no fault-free node will attempt to use the faulty component.

### Self-Checking Nodes

For all likely faults, a self-checking component must either produce the "correct" outputs or somehow indicate that the outputs are incorrect. A component that satisfies this requirement is said to be *fault secure*.<sup>10</sup> If the component is not guaranteed to produce an error indication immediately following the first fault, it is possible for several faults to exist in the component simultaneously without any indication to the rest of the system. Even if the component is fault secure with respect to any single fault, several faults together may lead to the failure of the self-check mechanism and, eventually, to incorrect outputs from the component being accepted as correct by the rest of the system. In order to prevent this situation, the component must be *self-testing*.<sup>10</sup> In the presence of one or more faults, a self-testing component produces an error indication before additional faults can occur and lead to the failure of the self-check mechanism. Components which are fault-secure and self-testing are said to be *totally self-checking*.<sup>10</sup> (TSC).

Error detecting/correcting codes can be used to implement TSC nodes. Redundant information is carried by busses, memories, and registers in order to detect (and possibly correct) errors.<sup>10</sup> Unfortunately, different coding schemes must be used for different parts of the node. The resulting increase in the complexity of the design and of design verification and testing may lead to a circuit in which failure modes that are more difficult to predict and "tolerate" are more likely to occur.

An alternative is to construct the TSC computation or communication node using two identical, *independent* modules, each performing the function of the node. Inputs from neighbor nodes are fed to both modules. If the modules operate synchronously, their outputs should always be identical. Except for the nearly-impossible case where both modules produce identical *incorrect* output, an error can be detected by a comparator which is part of the node. The output of the comparator is connected to neighboring nodes through dedicated wires. The output from one of the two modules is the "functional" output from the node (Fig. 1). A "no-match" signal from the comparator is used locally as a reset signal and is also sent to all neighbors as a failure indicator. Similar failure indicators from the neighbors cause an interrupt and invoke system-level routines that handle the node failure.

Implementing the TSC property in a component using duplication and matching may appear wasteful since it more than doubles the required hardware. However, this scheme becomes more attractive when issues such as design complexity, fault coverage,

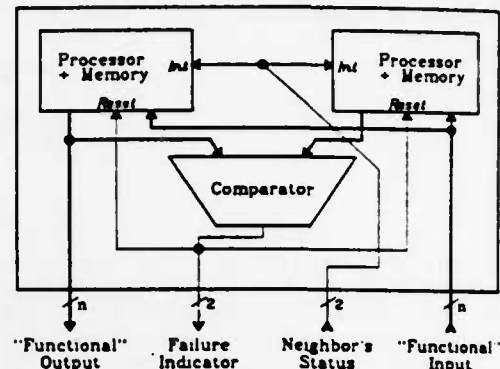


Fig. 1: A Self-Checking Computation Node

reliability prediction, and the ability to recover from transient faults are taken into account. Traditional fault models are not valid for VLSI.<sup>12</sup> As a result, low-cost error detection schemes, which are based on these models, may no longer be adequate. With duplication and matching, errors are detected as long as the comparator remains functional and the two modules produce different outputs the first time one or both of them fail. Since a faulty comparator can mask faulty functional modules, faults in the comparator must not go undetected, i.e., the comparator must be self-testing. Thus a detailed analysis of the effects of all likely faults on the comparator is required.

### Physical Defects and Logical Faults in VLSI

The design of self-checking circuits requires an understanding of the physical defects which commonly occur in VLSI and of the resulting logical faults. In the past the stuck-at fault model has been widely used as model, at the logical level, the effects of physical defects in circuits. This model does not cover many of the possible defects in VLSI.<sup>3,5,12</sup> The fabrication flaws and physical processes that can cause malfunction of NMOS and CMOS VLSI circuits are summarized in this section.

VLSI chip failures may be caused by design or fabrication flaws, may be due entirely to environmental factors, or are the end result of a degenerative process due to operational and environmental stresses but partially attributable to design or manufacturing defects.<sup>4,10</sup> Fabrication defects in MOS chips consist mainly of shorts and opens in each interconnection level, shorts between different levels, and large imperfections such as scratches across the chip.<sup>5</sup> Other fabrication defects include incorrect dosage of ion implants, contact windows that fail to open, misplaced or defective bonds, and penetration of the package by contaminants.<sup>4</sup> During the operation of the chip, faults may be caused by electromigration, corrosion, electrical breakdown of oxide, cracks due to thermal expansion, power supply fluctuation, and ionizing or electromagnetic radiation.<sup>4</sup>

At the logical level, most of the faults can be represented in a circuit model that consists of a network of switches, loads (for NMOS), and interconnection lines which directly correspond to the transistors and interconnections in the actual circuit.<sup>5</sup> Most of the physical defects, such as opens and shorts, can be represented in this model in an obvious way.<sup>3</sup> A "switch" may be permanently on or permanently off corresponding to a gate input stuck-at-1 or 0, respectively. Shorted NMOS loads (pullups) are equivalent to an output line s-a-1. Disconnected gate inputs are usually equivalent to s-a-0 or s-a-1 faults.

Some physical defects have a more complex effect on the circuit. In NMOS, incorrect dosage of ion implants may cause a threshold shift in a load transistor. This can result in an output voltage that lies between the voltages assigned to logic 0 and logic 1. If the fanout from the gate is greater than one, some of the gates connected to its output may "interpret" it as logic 1 while others will interpret it as logic 0. If, at some point in time (clock cycle), the line is supposed to be a logic 1 but is interpreted by some of the gates as logic 0, we call it a *weak 1* fault. Conversely, if the line is supposed to be a logic 0 but is interpreted by some of the gates as logic 1, we call it a *weak 0* fault. A single physical defect, resulting in a single weak 0 or weak 1 fault, has the same effect as multiple s-a-1 or s-a-0 faults, respectively.

In CMOS, a transistor which is permanently off or a break in a line can result in a high impedance state where the output of a combinational logic gate is dependent on the previous output rather than the current input.<sup>12</sup> Such a fault (called a *stuck-open* fault) may escape detection even if all possible input vectors are used to test the circuit.<sup>12</sup>

#### Implementation of Self-Testing Comparators in VLSI

The duplication and matching scheme relies entirely on a self-testing comparator to detect faults in the functional modules. Implementing such a comparator requires knowledge of how different faults will affect the circuit. Fortunately, a comparator is a simple circuit that can be implemented with a regular structure and is therefore amenable to thorough analysis. Hence, we can have confidence in our ability to predict the likely physical defects, develop a valid fault model, and prove that the implementation we propose is indeed self-testing.

We assume that physical defects in the node occur one at a time. A fault that is the result of a single physical defect is called a *single fault*. It is assumed that there is a negligible probability that the time interval between the occurrence of successive single defects in the comparator or between a single defect in the comparator and an arbitrary collection of defects in the functional modules, is less than some value  $T$ . In order to ensure that faults in the comparator will not mask future faults in the functional units, during normal operation, the comparator must "test itself" for any single fault in less than time  $T$ .

#### Single Stuck-At Faults

As a first step to constructing a comparator which is self-testing with respect to any single fault, we will discuss the implementation of a comparator which is self-testing with respect to any single *stuck-at* fault.

In this context "two-rail" codes prove useful. They consist of all words (bit vectors) such that a specified half of the word is the complement of the other half. If the output of one of the modules in a self-checking node is complemented, a two-rail code checker can serve as a "comparator" that checks the validity of the output. Such a checker, which is self-testing with respect to any single stuck-at fault, can be implemented as a two level NOR-NOR PLA (Fig. 2).<sup>3,2</sup> The output from the checker is a two-bit two-rail code that is 01 or 10 (*code output*) if the input is a two-rail code word (*code input*), and 00 or 11 (*noncode output*) otherwise (*noncode input*). It can be shown that if any single stuck-at fault exists in the checker, there is an input two-rail code word that results in a 00 or 11 output, thereby "detecting" the fault.<sup>13</sup>

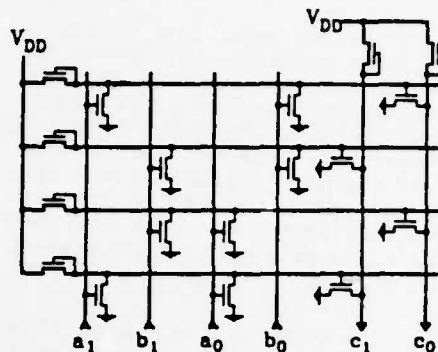


Fig. 2: An NMOS Self-Testing Two-Rail Code Checker

The requirement that the checker must be self-testing with respect to any single stuck-at fault poses severe constraints on its implementation. It can be shown that any two level AND-OR (or NOR-NOR) implementation for an input of  $2n$  bits ( $n$  bits from each module) must use  $2^n$  product terms, one for each code input.<sup>11</sup> If the output from each module is, say, 18 bits, this implementation is impractical since it requires  $2^{18} = 85536$  product terms. Furthermore, all possible ( $2^n$ ) code words must appear at the checker's inputs for it to perform a complete self-test.

Several small self-testing two-rail code checkers can be used as "cells" for constructing a self-testing checker for a wide input word (Fig. 3).<sup>10,6</sup> While the self-testing property is preserved, the number of input patterns required for a complete self-test is dependent only on the size of the largest "cell".<sup>6</sup>

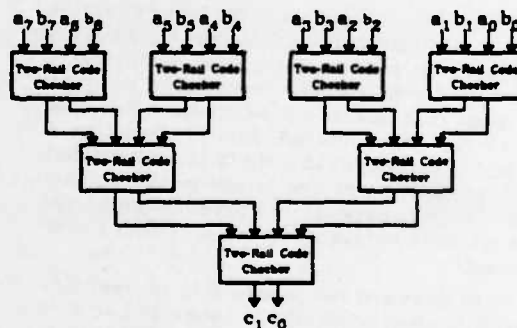


Fig. 3: A Self-Testing Two-Rail Code Checker Tree

#### Other Single Faults

The faults that commonly occur in a MOS PLA are stuck-at faults, shorts between adjacent lines, breaks in lines, and contact faults which include missing or extra devices at crosspoints.<sup>3,7</sup> In addition, weak 0/1 faults can occur on the input or product term lines. Fortunately, it turns out that a straightforward NOR-NOR PLA implementation of the checker discussed above is self-testing with respect to any one of the aforementioned single faults. The rest of this section contains an informal "proof" of this claim; a more formal proof will be presented elsewhere.<sup>11</sup> Faults in the input lines, product term lines, output lines, AND array crosspoints, and OR array crosspoints, are considered separately.

Any single stuck-at fault or short in the input lines will cause one or more 0's to change to 1's or one or more 1's to change to 0's (but not both) for some code input. It can be shown that such an error (called a

unidirectional error<sup>7)</sup> on the input lines results in noncode output.<sup>13</sup> A break in the input line outside the AND array is equivalent to the line stuck-at-0 or stuck-at-1. A break in the middle of the AND array affects only some product terms. For an affected product term, if the break is equivalent to a stuck-at-1, the one code input that is supposed to select this product term won't, and a noncode output will result. If the break is equivalent to a stuck-at-0, there exists a code input that results in a noncode output since it selects two product term lines each of which is connected to a different output line.<sup>11</sup>

An extra device in the AND array is equivalent to the corresponding product term stuck-at-0. The code input that is supposed to select that product term results in a noncode output. If there is a missing device in the AND array, there exists a code input that produces a noncode output since it selects two product term lines, each of which is connected to a different output line.<sup>11</sup>

An extra device in the OR array means that one of the product terms is connected to both outputs. A missing device in the OR array is equivalent to the corresponding product term stuck-at-0. In either case, the code input that selects the relevant product term will result in a noncode output.

If the output lines are shorted, their values are equal and that is a noncode output. If one of the lines has a stuck-at fault, there exists a code input that causes the other line to have the same value so the output is noncode. For some code input, a break in one of the output lines is equivalent to a stuck-at-1 or stuck-at-0 fault on that line.

A stuck-at-0 fault on a product term line will result in a noncode output if the input is the code word that is supposed to select that product term line. A stuck-at-1 fault on a product term line will result in a noncode output to any input that selects a product term line that is connected to the other output line. A break in a product term line is equivalent to a stuck-at fault on that line since each product term line is connected to only one output line. A short between two product term lines will result in a noncode output if the input selects either one of these lines.<sup>11</sup>

Product term lines are not susceptible to weak 0/1 faults since each product term line is connected to only one output line (fanout of one) so that a weak 0/1 fault is equivalent to a single stuck-at fault. Input lines have a fanout greater than one and are thus susceptible to weak 0/1 faults. A weak 1 fault on an input line is equivalent to one or more missing devices in the AND array. Each product term that is connected to a "missing device" will be selected by an input code word that also selects a product term line that is connected to the other output line.<sup>11</sup> Thus, a noncode output will result. A weak 0 fault on an input line is equivalent to one or more product term lines which are stuck-at-0. Any code input that is supposed to select one of these product terms will result in a noncode output.

In CMOS chips PLAs are usually implemented in dynamic "pseudo NMOS."<sup>12</sup> All product term and output lines are precharged during every clock cycle before being selectively discharged according to the input. Therefore no state is preserved from one cycle to the next and the circuit is combinational despite any opens in the precharge or discharge paths.<sup>11</sup> Hence the PLA used in CMOS chips is only susceptible to the same faults as the traditional static PLA used in NMOS chips.

This analysis shows that for all single faults in our fault model, there exists a code input that results in a noncode output from the proposed two-rail code checker PLA. Thus, the checker is self-testing with respect to any likely single fault. Based on this result, it can be shown that the checker constructed as a tree of smaller self-testing checkers (Fig. 3) is also self-testing with respect to any likely single fault.<sup>11</sup>

### Summary and Conclusions

We have presented an approach to increasing the reliability of future high-end systems beyond what is possible with technological solutions alone. The system consists of computation nodes and communication nodes, interconnected by high-speed dedicated links. These components are relied upon to detect errors while system level protocols are used for error recovery and reconfiguration.

The use of duplication and matching for implementing the self-checking nodes allows us to restrict a detailed analysis of the impact of all possible faults to the comparator, which is a relatively simple circuit. We have shown that the self-testing comparator, which is the backbone of our approach, can be implemented with NMOS and CMOS technologies.

### Acknowledgements

We would like to thank Richard Fujimoto, Manolis Katevenis, and Robert Sherburne for reviewing a draft of this paper.

This research was supported by the State of California MICRO program and the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

### References

1. T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *FTCS12*, pp. 29-33 (June 1982).
2. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *IFIPS Proc.*, pp. 878-883 (August 1968).
3. B. Courtois, "Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits," pp. 341-350 in *VLSI 81*, ed. J. Gray, Academic Press (1981).
4. E. A. Doyle, "How Parts Fail," *IEEE Spectrum* 18(10) pp. 36-43 (October 1981).
5. J. Gallay, Y. Crouzet, and M. Vergnault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE TC C-29*(6) pp. 627-631 (June 1980).
6. J. Khakbaz and E. J. McCluskey, "Concurrent Error Detection and Testing for Large PLA's," *IEEE JSSC* 17(2) pp. 386-394 (April 1982).
7. G. P. Mak, J. A. Abreham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *FTCS12*, pp. 303-310 (June 1982).
8. R. A. Rasmussen, "Automated Testing of LSI," *Computer* 15(3) pp. 69-78 (March 1982).
9. C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," in *Proc. Advanced Course on VLSI Architecture*, Univ. of Bristol, England, ed. P. C. Treleaven, Prentice Hall (1982).
10. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).
11. Y. Tamir, "Fault Tolerance for VLSI Multicomputers," Ph.D. Dissertation (in preparation).
12. R. L. Wedeck, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *BSTJ* 57(5) pp. 1446-1474 (May-June 1978).
13. S. L. Weng and A. Avizienis, "The Design of Totally Self-Checking Circuits Using Programmable Logic Arrays," *FTCS9*, pp. 173-180 (June 1979).

## Design and Application of Self-Testing Comparators Implemented with MOS PLAs

*Yuval Tamir and Carlo H. Séquin*

Computer Science Division  
Department of Electrical Engineering and Computer Sciences  
University of California  
Berkeley, California 94720

### Abstract

A high probability of detecting errors caused by hardware faults is an essential property of any fault-tolerant system. VLSI technology makes the use of *duplication and matching* for error detection practical and attractive. A critical circuit in this context is a *self-testing comparator*. Faults in the comparator must be detected so that they do not mask discrepancies between the duplicated modules.

This paper discusses the implementation of comparators which are self-testing with respect to faults caused by any single physical defect likely to occur in NMOS and CMOS integrated circuits. A new fault model for PLAs is presented. This model reflects several physical defects in VLSI circuits which are not accounted for in previously published models. It is shown that in a self-testing comparator, implemented as a single two-level NOR-NOR PLA, the number of required product terms grows *exponentially* with the number of input bits. A particular design of a comparator using a single two-level NOR-NOR PLA is discussed. The operation of this comparator under the faults in the fault model is analyzed in detail. The comparator is proven to be self-testing with respect to any likely single fault in the proposed fault model, provided that several guidelines about its physical layout are followed. The use of this comparator as a basic building block of fault-tolerant systems is discussed.

December 1983



## I. INTRODUCTION

Ideally, a computer system must always generate the "correct" results. Since real systems suffer from design faults, fabrication defects, and other hardware faults, this ideal can never be reached. A minimum requirement from any computer system is that it must not mislead the "outside world" into accepting incorrect outputs as correct. Hence, the "outside world" must be notified (or be able to detect) when errors occur. *Fault-tolerant* systems attempt to achieve a higher probability of producing correct outputs by adapting to changes in the system caused by faults, and continuing correct operation. Explicit or implicit *error detection* is not only required for preventing acceptance of incorrect outputs but is also a necessary first step of any scheme for "recovering" from faults [2].

*Checker* circuits play a key role in systems with *on-line* error detection. They continuously verify that certain sets of lines in the system carry values that together conform to some *code*. Some of the typical codes used are: parity codes, m-out-of-n codes, arithmetic codes, and two-rail codes [28]. When such codes are used, it is assumed that faults will cause the values on the monitored lines to change in such a way that they will no longer conform to the code. If faults modify the values in unexpected ways so that the incorrect values still conform to the code, the error cannot be detected.

In modern VLSI chips the traditional fault model, that takes into account only single stuck-at faults, is no longer valid [13,27]. For example, a single physical defect may result in erroneous values on several lines or may convert a combinational circuit into a finite state machine. Hence, simple error-detecting codes, such as a single parity bit, may fail to detect many of the possible errors. Furthermore, evaluating the percentage of faults that can be detected by a given scheme (fault-coverage) is very difficult since we cannot use the traditional method of simulating the effects of all possible faults [20].

No single type of error-detecting code is suitable for use throughout a complex chip such as a microprocessor. While Hamming codes may be used for detecting errors in registers or bus transfers [28], arithmetic codes [3] are needed for checking the operation of the ALU, and duplication and matching must be used for checking control lines and logical operations [8,26]. The need to use different codes and implement different checkers in different parts of the chip exacerbates the already difficult problem of managing the complexity of VLSI chip design [21]. The increase in the

complexity of the design and of its verification decreases the confidence that the design is correct, thereby reducing the overall reliability of the system.

Modern VLSI technology makes duplication and matching an attractive and economically feasible way of implementing error detection. Duplicate high-level functional modules, such as microprocessors or communication chips, can run in parallel, and errors can be detected by comparing module outputs[10,22,24]. Errors are detected as long as the first time one, or both, of the two modules fail(s), they produce different outputs. There is no need to predict exactly how defects will affect the modules and there is no increase in the complexity of designing, verifying the design, and testing the chips.

Duplication and matching may also be used to alleviate the problem of system failure due to chips with fabrication defects or undetected design faults. Separately designed functional modules can be used in order to detect latent design and fabrication faults during the operation of the system and prevent undetected incorrect outputs[4]. Using different designs also virtually eliminates the possibility that the two modules fail in exactly the same way at exactly the same time.

The critical element in any duplication and matching scheme is the circuit that compares the outputs from the two functional modules. Undetected faults in this comparator can mask discrepancies between the outputs of the functional modules. Hence, the comparator must be *self-testing*[1], i.e., during normal operation physical defects in the comparator must result in an error indication.

This paper discusses the design and implementation of self-testing comparators in VLSI. It focuses on designs based on programmable logic arrays (PLAs)[18]. Large VLSI chips are far too complex to allow detailed analysis of all the possible physical defects that can occur and of the effects of these defects on the operation of the circuit. On the other hand, PLAs are characterized by a simple regular structure and are therefore more amenable to thorough analysis. Based on such analysis, a new fault model for PLAs is developed. The model reflects some physical defects that are likely to occur in integrated circuits but are not taken into account in previously published models.

Comparators implemented with two-level AND-OR or NOR-NOR circuits, which are claimed to be self-testing, have been presented in the literature[6,29]. We show that these designs, which require that the number of product terms grow exponentially with

the number of input bits, are *optimal* in terms of size. We present a formal proof that a comparator implemented as a NOR-NOR PLA, based on these designs, is self-testing with respect to most single faults in the new fault model. We propose a few simple layout guidelines that help ensure that the comparator is self-testing with respect to the remaining faults.

Finally, we discuss the application of the self-testing comparator as a basic building block for implementing fault-tolerant systems.

## II. THE FAULT MODEL

In order to design and implement self-testing circuits, it is necessary to consider the physical defects that are likely to occur with the specific technology being used. However, the design is done at the level of boolean logic ("ones and zeros") rather than at the level of voltages, currents, and charges. Hence, once the likely physical defects are known, it is desirable to determine the effects that these defects have on the operation of the circuit at the logical level. A description of these effects is called a *fault model*. In this section we present a fault model for general NMOS and CMOS VLSI circuits and use it to develop a detailed fault model for PLAs.

### A. *Faults in MOS VLSI Digital Circuits*

The failure of a VLSI chip may be due to design or fabrication flaws, environmental factors, or a combination of the two[11,14]. The resulting physical defects consist mainly of breaks in lines, shorts between lines at the same interconnection level (metallization, diffusion, and poly-silicon), shorts through the insulator separating different levels, shorts to the substrate, and large imperfections such as scratches across the chip[9,13]. Other possible defects are incorrect dosage of ion implants, contact windows that fail to open, and misplaced or defective bonds[11]. During the operation of the chip, faults may also be caused by power supply fluctuation, and ionizing or electromagnetic radiation[7, 11].

While the stuck-at fault model[12] can represent the effects of a significant percentage of the physical defects that occur in modern NMOS and CMOS VLSI circuits, it cannot represent the effects of several other possible defects and is therefore insufficient[9,13,27]. The effects of most defects can be represented, at the logical level, by a circuit model that consists of a network of switches, loads (for NMOS), and



interconnection lines which directly correspond to the transistors and interconnections in the actual circuit [13]. Shorts and breaks in lines can be represented with this circuit model in an obvious way [9]. Shorts to "ground" and "power" are the traditional stuck-at faults. A "switch" may be permanently on or permanently off, corresponding to a gate input stuck-at-1 or 0, respectively. Shorted NMOS loads (pull-ups) are equivalent to an output line s-a-1. Disconnected gate inputs are usually equivalent to s-a-0 or s-a-1 faults. A single break in a line that fans out to many inputs is equivalent to multiple stuck-at faults (all of the same type).

Some physical defects have a more complex effect on the circuit. In NMOS, incorrect dosage of ion implants may cause a threshold shift in a load transistor. This can result in an output voltage that lies between the voltages assigned to logic 0 and logic 1. If the fanout from the gate is greater than one, some of the attached gates may "interpret" its output as logic 1 while others will interpret it as logic 0. If, at some point in time (clock cycle), the line is supposed to be a logic 1 but is interpreted by at least one of the gates as logic 0, we call it a *weak 1* fault. Conversely, if the line is supposed to be a logic 0 but is interpreted by at least one of the gates as logic 1, we call it a *weak 0* fault [24]. It is clear that a line may exhibit both a weak 0 fault and a weak 1 fault, as a result of a single physical defect.

A stuck-at-1 fault is a degenerate case of a weak 0 fault while a stuck-at-0 fault is a degenerate case of a weak 1 fault. If a line is stuck-at-1, *all* the devices connected to it *always* interpret its value as logic 1. If a line has a weak 0 fault, *at least one* of the devices connected to it *always* interprets it as a logic 1.

Breaks in lines are another possible source of weak 0 and weak 1 faults. A break may result in a segment of the line that is only connected to gates of MOS transistors and is therefore essentially "floating." The gates connected to the floating segment of the line receive an incorrect value for the line in one of its states (0 or 1).

A single break in the line can result in the line being stuck-at-1 if all the pull-down devices are disconnected from the rest of the line, and in the line s-a-0 if all the pull-up (or load) devices are disconnected from the rest of the line. Furthermore, if only some of the pull-up or pull-down devices are disconnected from the line, the line may not be s-a-0 or s-a-1 but assume the wrong value for some inputs that only turn on the disconnected pull-ups or pull-downs. In the worst case, in CMOS or dynamic logic circuits, a break in a

line or a transistor that is permanently off can make the output of a supposedly combinational logic circuit dependent on the previous output rather than the current input alone [27]. Such a fault (called a *stuck-open* fault) may escape detection even if all possible input vectors are used to test the circuit [27].

A short between adjacent or crossing lines forces both lines to have the same value. This value may lie between logic 0 and logic 1. Hence, if the two lines are supposed to carry complementary values, the line that is supposed to be at logic 1 may have a weak 1 fault and the line that is supposed to be at logic 0 may have a weak 0 fault. If the circuit is designed so that a short always forces both lines to a well defined logic value, this value may be always the value of one of the lines that "dominates" because it is driven with larger devices, or it may always be logic 0 (AND operation) or always logic 1 (OR operation).

We assume that if the fault is *transient*, the circuit returns to its original physical structure after the fault has disappeared. It is, of course, possible for a transient fault to cause a permanent change in the *state* of a circuit with memory elements. We assume that, for the duration of the fault (defect), the effects of the defect are *deterministic* so that under identical conditions the effects of a particular defect are always the same. Thus, if a line has a weak 1 fault due to its driver, the devices connected to it which misinterpret the logic 1 as a logic 0, *always* misinterpret the logic 1 as a logic 0.

Traditionally, the term *single fault* has been used to denote an erroneous logic value on a single line in the circuit. From the above it is clear that a single physical defect may result in erroneous logic values on several lines in the circuit. Hence, we will use the term *single fault* to denote the effect, at the logical level, of a single physical defect.

#### *B. A Fault Model for MOS PLAs*

For any VLSI circuit, the effect of a single fault on the output is dependent on layout details such as which lines are adjacent, which lines cross each other, *etc.* One of the advantages of using PLAs is that their regular structure simplifies analysis of the effects of faults on its outputs and therefore facilitates test vector generation and determination of fault coverage. In this section we discuss how the faults discussed above affect the operation of a two-level NOR-NOR MOS PLA. To facilitate this discussion, a "typical" NMOS PLA is shown in Fig. 1.

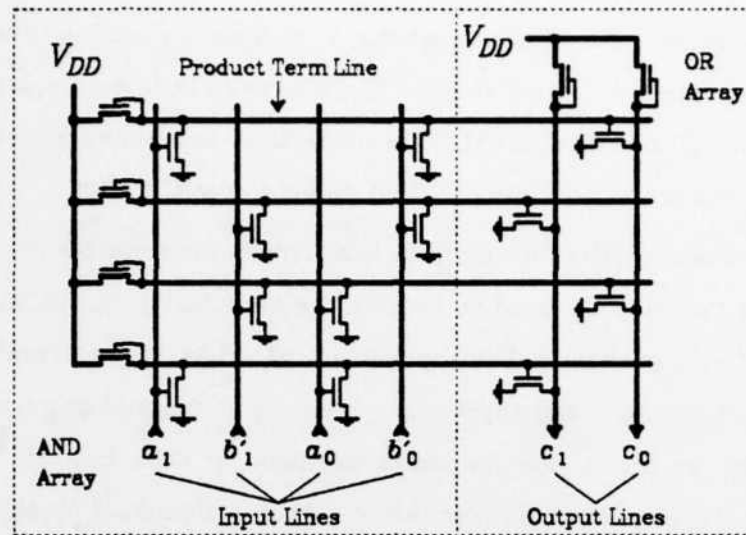


Fig. 1: A Self-Testing NMOS Two-Rail Code Checker

The most elementary fault model used for PLAs includes three types of faults[16, 19, 29]:

- (I) A stuck-at fault on an input line, product term line, or output line.
- (II) A short between two adjacent or crossing lines that forces both of them to the same logic value.
- (III) A missing or extra crosspoint device in the AND array or in the OR array.

The first two types of faults were explained above and correspond directly to physical defects in the circuit. The third type of faults refers to faults whose effect on the operation of the circuit is equivalent to the effect of a missing or extra crosspoint device. This may be the result of the gate of the crosspoint device stuck-at its "off" value (0 for NMOS, 1 for PMOS) when it should be connected to an input or product term line, or connected to an input or product term line when, by design, it should be permanently held at its "off" value.

A missing crosspoint device has the same effect as a device that always misinterprets the line that drives it as a logic 0 even when it is a logic 1. Thus, a missing crosspoint device fault in the AND array is equivalent to a weak 1 fault on the corresponding input line while a missing crosspoint device fault in the OR array is equivalent to a weak 1 fault on the corresponding product term line. Hence, if weak 1 faults on input lines and product term lines are considered, there is no need to consider

missing crosspoint device faults separately.

The above three fault types do not include weak 0 and weak 1 faults or breaks in lines that are not equivalent to stuck faults. Since breaks in lines are one of the main causes of failures in VLSI circuits [9, 13], it is clear that the above simple fault model does not accurately reflect the possible physical defects in a MOS PLA.

Some of the effects of breaks on general MOS circuits cannot occur in PLAs due to their structure. This fact can be used to reduce the complexity of the fault model that must be considered in analyzing the operation of PLAs under faults. One such simplification relies on the fact that input lines are only connected to gates of devices in the PLA. A break in an input line causes a segment of that line to "float" and is therefore equivalent to a weak 0 and/or weak 1 fault. Hence, if weak 0/1 faults on inputs lines are taken into account, there is no need to consider breaks in input lines separately.

Another important simplification of the fault model is based on the fact that product term lines and output lines only have one pull-up (load) device and that this device is independent of the inputs to the circuit. Every point in a product term or output line is either connected to the single pull-up (load) or permanently disconnected from it (due to a break). For any input, segments of the line that are connected to the pull-up are either set to logic 1 or set to logic 0 by some pull-down device that is turned on by that particular input. A segment of the line that is disconnected from the pull-up is set to logic 0 by the first input that is supposed to set it to 0 and stays stuck-at-0 for a long time thereafter. Hence no state is preserved on lines between inputs (clock phases). The troublesome faults that can convert a general combinational circuits into a sequential circuit cannot occur.

Based on the above discussion, a realistic fault model for PLAs must include weak 0/1 faults as well as the possible effects of breaks in product term lines and output lines. Specifically, the following faults must be considered:

- (A) Weak 0 or weak 1 or both on one input line.
- (B) A short between two adjacent input lines.
- (C) Weak 0 or weak 1 or both on one product term line.
- (D) A short between two adjacent product term lines.
- (E) Weak 0 or weak 1 or both on one output line.

- (F) A short between two adjacent output lines.
- (G) A short between an input line and a crossing product term line.
- (H) A short between a product term line and a crossing output line.
- (I) An extra crosspoint device in the AND array.
- (J) An extra crosspoint device in the OR array.
- (K) A break in a product term line.
- (L) A break in an output line.

### III. BACKGROUND AND TERMINOLOGY

Since code checkers are key elements in computer systems with on-line error detection, the design and implementation of various self-testing checkers has been an active research area for many years. This section includes a discussion of some of that work that is relevant to self-testing comparators in VLSI. In addition, the terminology and notation that will be used in the rest of the paper are introduced.

We will assume that two  $n$ -bit vectors,  $A = (a_{n-1}, a_{n-2}, \dots, a_0)$  and  $B = (b_{n-1}, b_{n-2}, \dots, b_0)$ , are to be compared. In much of the literature *two-rail code checkers* rather than *comparators* are discussed. Given two  $n$ -bit vectors  $X = (x_{n-1}, x_{n-2}, \dots, x_0)$  and  $Y = (y_{n-1}, y_{n-2}, \dots, y_0)$ , the combined  $2n$  bit vector  $XY = (x_{n-1}, \dots, x_0, y_{n-1}, \dots, y_0)$  is a two-rail code word if  $x_i = y'_i$  for all  $i$  such that  $0 \leq i \leq n-1$  (where  $y'_i$  means the complement of  $y_i$ ). We will use  $B'$  to denote an  $n$ -bit vector whose elements are the complements of the elements of  $B$ , i.e.,  $B' = (b'_{n-1}, b'_{n-2}, \dots, b'_0)$ . A two-rail code checker whose input is the bit vector  $AB'$  is, effectively, a comparator of vectors  $A$  and  $B$ . We will start out by making the assumption (that will later be shown to be unnecessary) that all the input bits are available in both complemented and uncomplemented form. Hence there is no difference between the design of comparators and two-rail code checkers; and we will use the terms "comparator" and "two-rail code checker" interchangeably.

For a given *fault set*  $F$ , a checker is said to be *self-testing* if for every fault  $f \in F$  there is a code input that results in a noncode output (error indication)[1]. When duplication and matching is used for on-line error detection, we assume that faults (both *permanent* and *transient*) do not occur simultaneously in the duplicated functional modules and in the comparator. If the first fault occurs in one of the functional modules and causes a permanent change in the circuit or in the state of the circuit, we assume



that the outputs from the two modules "disagree" and result in a noncode output from the comparator before any faults occur in the comparator. If the first fault occurs in the comparator, we assume that, before additional faults can occur in the comparator or the functional modules, the set of code words that will appear as inputs to the comparator will be sufficient to achieve a complete self-test of the comparator. Depending on the particular implementation of the comparator, this set of code words may have to include all possible code inputs. If the fault persists for this duration and the comparator is self-testing, its output will indicate an error before a fault in one of the functional modules can lead to an undetected erroneous *functional* output from the system. Based on these assumptions, the comparator only needs to be self-testing with respect to a fault set  $F$  that includes all *single faults*, as defined in Section II.

Pioneering work in the field of self-testing checkers was reported by Carter and Schneider [6] whose design of a self-testing two-rail code checker serves as a basis for the comparator discussed in this paper. For the case  $n = 2$ , Carter and Schneider presented a design of a circuit that checks whether the input is a two-rail code word and that is also self-testing with respect to any single stuck-at fault [6]. The circuit, shown in Fig. 2, has two output lines  $c_1$  and  $c_0$  where  $(c_1, c_0) = (0, 1)$  or  $(c_1, c_0) = (1, 0)$  for code input, and  $(c_1, c_0) = (0, 0)$  or  $(c_1, c_0) = (1, 1)$  for noncode input.

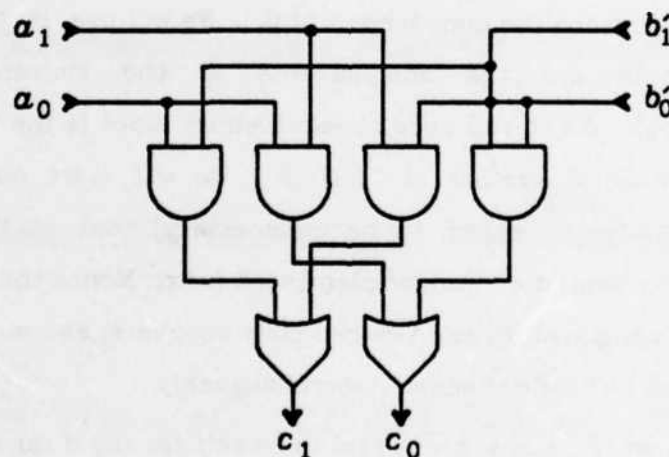


Fig. 2: A Self-Testing Two-Rail Code Checker [6]

Carter and Schneider's checker has the property that, with no faults, every line in the circuit is 0 for at least one code input and 1 for at least one code input. If any line is stuck-at-0 (s-a-0) or s-a-1, the code input for which the line is supposed to be at 1 or 0, respectively, results in the output  $(0, 0)$  or  $(1, 1)$ .

Wang and Avizienis [29] extended Carter and Schneider's design to arbitrary size input vectors. For each one of the  $2^n$  input code words there is a single unique product term that is selected (set to 1) only by that code word. Each product term line selects exactly one of the two output lines. Depending on the parity of the vector  $A = (a_{n-1}, a_{n-2}, \dots, a_0)$ , half the code inputs select output  $c_0$  and the other half select output  $c_1$ .

The checker proposed by Wang and Avizienis can be described by sum-of-products equations as follows: For any integer  $k$ , let  $I_k$  denote the set of the  $k$  integers between 0 and  $k-1$ , i.e.  $I_k = \{0, 1, \dots, k-2, k-1\}$ . If  $Q$  is a set, let  $|Q|$  denote the number of elements in  $Q$ .

$$\begin{aligned} c_0 &= \sum_{\{Q | Q \subset I_n \text{ and } |Q| \text{ even}\}} \left\{ \left( \prod_{i \in Q} a_i \right) \left( \prod_{j \in (I_n - Q)} b'_j \right) \right\} \\ c_1 &= \sum_{\{Q | Q \subset I_n \text{ and } |Q| \text{ odd}\}} \left\{ \left( \prod_{i \in Q} a_i \right) \left( \prod_{j \in (I_n - Q)} b'_j \right) \right\} \end{aligned} \quad (1)$$

In NOR-NOR form, similar functionality can be achieved based on the Equation (2). An NMOS PLA which implements these equations for the case  $n = 2$  is shown in Fig. 1. It should be noted that there are a total of  $2n$  input bits to this circuit: all the "a" bits uncomplemented and all the "b" bits complemented. Each "product term" contains exactly  $n$  literals.

$$\begin{aligned} c_0 &= \sum_{\{Q | Q \subset I_n \text{ and } |Q| \text{ odd}\}} \left\{ \text{NOR} \left\{ \{a_i | i \in Q\} \cup \{b'_j | j \in (I_n - Q)\} \right\} \right\} \\ c_1 &= \sum_{\{Q | Q \subset I_n \text{ and } |Q| \text{ even}\}} \left\{ \text{NOR} \left\{ \{a_i | i \in Q\} \cup \{b'_j | j \in (I_n - Q)\} \right\} \right\} \end{aligned} \quad (2)$$

In Section V it is shown that the circuit described by Equation (2) is, in fact, a comparator. In Sections VI and VII it is shown that the circuit can be implemented so that it is self-testing with respect to the fault model presented in Section II.

In the literature, checkers which are *fault-secure* as well as self-testing have often been discussed [1, 17]. A circuit is said to be fault-secure if, for every fault in the prescribed fault set, the circuit never produces an incorrect code output for code



inputs[1]. A checker only provides one bit of information to the rest of the system regarding its inputs; the checker's output is code or noncode depending on whether its input is code or noncode, respectively. As long as this binary distinction is performed correctly, it does no matter exactly which code output (or which noncode output) is produced by the checker. Thus, the concept of fault-secure checkers is meaningless[23] and the fault-secure property will not be considered further in this paper.

#### IV. OPTIMAL DESIGN OF SELF-TESTING COMPARATORS USING TWO-LEVEL LOGIC

Published work on self-testing checkers usually consists of a circuit design and a proof that the circuit is self-testing. There has been no attempt to show that the proposed designs are optimal in any respect. The self-testing comparator design proposed by Wang and Avizienis requires  $2^n$  product terms for comparing  $n$ -bit vectors. However, it is possible to implement a comparator that has two outputs that are (0,1) or (1,0) for code inputs and (1,1) for noncode inputs based on the equations:

$$c_0 = a_0' + b_0' + \sum_{i=1}^{n-1} (a_i b_i' + a_i' b_i) \quad c_1 = a_0 + b_0 + \sum_{i=1}^{n-1} (a_i b_i' + a_i' b_i)$$

This comparator is self-testing with respect to faults in the input lines and output lines but requires only  $4n$  product terms. However, this comparator is *not* self-testing with respect to stuck faults on the product term lines. The question thus arises whether  $2^n$  is the minimum number of product terms necessary for a comparator that is self-testing with respect to a realistic fault model which also takes into account faults affecting the product term lines.

Since the comparator must be self-testing with respect to stuck-at faults on the output lines, it must have at least two output lines[6]. The use of more than two lines has been proposed[23]; however, since limited communication bandwidth is a severe problem in VLSI systems, it is preferable to minimize the bandwidth dedicated to transmitting self-testing information. Hence we will only consider comparators with two output lines.

There are two possible ways to "code" the output from the comparator and still allow self-testing of the output lines: (A) The code output is (0,1) or (1,0) and the noncode (error indication) output is (0,0) or (1,1). (B) The code output is (0,0) or (1,1) and the noncode (error indication) output is (0,1) or (1,0). Option (A) is preferable since it allows the comparator to be self-testing with respect to shorts between the output

lines as well as any other fault that causes a *unidirectional error*. A unidirectional error means that, due to a fault, some lines that are supposed to be at logic 0 are at logic 1 or some of the lines that are supposed to be at logic 1 are at logic 0, *but not both*. It has been shown that the faults that are most likely to occur in PLAs (fault types (I), (II), and (III) in Section II), can result only in a unidirectional error[16]. Therefore only comparators with the option (A) encoding of the outputs will be considered.

Although the design of a self-testing comparator presented by Wang and Avizienis[29] uses  $2^n$  product terms, one for each code input, this is never shown to be a necessary property of self-testing comparators implemented with PLAs. In several papers[15,29] it is claimed that it is "desirable" to use PLAs that are *nonconcurrent*, i.e., where each code input selects only one product term. Wang and Avizienis propose a general approach to the design of self-testing PLAs that always results in a nonconcurrent circuit. They also give an example of a PLA where concurrency leads to a circuit which is not self-testing[29]. However, nonconcurrency is not presented as a *necessary* property nor is there any mention of a problem with product terms that are selected by more than one code input.

In this section it is shown that the exponential growth in the number of product term lines is indeed necessary for self-testing. For any two-level NOR-NOR implementation, it is shown that every code input must select exactly one product term line and that no two different code inputs can select the same product term line. This is necessary even if the only faults considered are single stuck-at faults on the input, output and product lines. The proof that the same requirement also applies to two-level AND-OR implementations is almost identical and will not be presented here.

**Lemma 1:** Every product term must be selected (set to 1) by least one code word.

**Proof:** Assume that there is a product term that is not selected by any code word. A stuck-at-0 fault on this product term line will not be detected during normal operation thus violating the self-testing requirement.

**Lemma 2:** Every code word must select at least one product term.

**Proof:** If there is some code word that does not select any product term, the comparator output for that code word will be the noncode output (1,1), which incorrectly signals an error.

**Lemma 3:** All the product terms selected by a single code word must be connected to

the same single output in the OR array.

*Proof:* If any of the product terms selected by a code word is connected to both outputs in the OR array, then, for that code word, the output will be the noncode output (0,0), which incorrectly signals an error. Similarly, the output will be (0,0) if the product terms are not all connected to the same output line.

**Lemma 4:** No product term can be selected by more than one code word.

*Proof:* By contradiction. Assume that  $P_i$  is a product term which is selected by the two code inputs  $AA = (a_{n-1}, \dots, a_0, a_{n-1}, \dots, a_0)$  and  $BB = (b_{n-1}, \dots, b_0, b_{n-1}, \dots, b_0)$ . Since the two code words are different, there exists an integer  $k$  ( $0 \leq k \leq n-1$ ) such that  $a_k \neq b_k$ .  $P_i$  is selected only if all the literals in the expression corresponding to  $P_i$  are 0. Since  $P_i$  is selected by both code words, it must be independent of bit  $k$  from the two functional modules. Hence  $P_i$  is also selected by the code input  $WW = (a_{n-1}, \dots, a'_k, \dots, a_0, a_{n-1}, \dots, a'_k, \dots, a_0)$  and by the noncode input  $Q = (a_{n-1}, \dots, a'_k, \dots, a_0, a_{n-1}, \dots, a_k, \dots, a_0)$ .

Since  $Q$  is a noncode input, the corresponding output produced by the comparator must be noncode. When  $P_i$  is selected, it sets to 0 the one output line it is connected to. Hence,  $Q$  must select another product term,  $P_j$ , connected to the other output line, so that the noncode output (0,0) will be produced. By Lemma 1,  $P_j$  must also be selected by at least one code word  $CC = (c_{n-1}, \dots, c_k, \dots, c_0, c_{n-1}, \dots, c_k, \dots, c_0)$ .

Since in  $CC$ , bit  $k$  from both functional modules is the same, and in  $Q$  bit  $k$  from one unit is the complement of bit  $k$  from the other unit,  $P_j$  must not include the literal corresponding to bit  $k$  from at least one of the two functional modules. Hence, since  $Q$  selects  $P_j$ , either  $AA$  or  $WW$  must also select  $P_j$ . Without loss of generality, assume  $WW$  selects  $P_j$ . From the above,  $WW$  also selects  $P_i$ . But in the OR array  $P_j$  is connected to a different output line from  $P_i$ . Hence, Lemma 3 above is "violated" and the code word  $WW$  results in the noncode output (0,0). Thus the assumption that there exists a product term that is selected by more than one code input must be incorrect.

**Lemma 5:** Every code word must select one, and only one, product term.

*Proof:* By Lemma 2, every code word must select at least one product term. Assume that the code word  $AA = (a_{n-1}, \dots, a_0, a_{n-1}, \dots, a_0)$  selects the two product terms  $P_i$  and  $P_j$ . By Lemma 4, no other code word except  $AA$  can select  $P_i$  or  $P_j$ . Hence, a stuck-at-0 fault on the  $P_i$  or  $P_j$  lines can only be detected by the input  $AA$ . By Lemma 3, both  $P_i$  and  $P_j$  must be connected to the same output line in the OR array.

Hence, when the code word  $AA$  is applied, the output from the PLA will be the same whether or not one of the product term lines  $P_i$  or  $P_j$  is stuck-at-0. Thus a stuck-at-0 fault on one of the product term lines  $P_i$  or  $P_j$  will not be detectable by any code word, thus violating the self-testing requirement.

**Theorem 1:** A self-testing comparator of two  $n$ -bit vectors that has two output lines and is implemented as a two level NOR-NOR PLA, must have *exactly*  $2^n$  product terms.

**Proof:** By Lemma 1, every product term line is selected by at least one code word. By Lemma 5, every code word selects one, and only one, product term line. Hence, the number of product term lines is equal to the number of code words. Since there are  $n$  bits of output from each one of the two functional modules, there are  $2^n$  code words. Therefore the number of product term lines is *exactly*  $2^n$ .

*Q.E.D.*

Any comparator of two  $n$ -bit vectors must have at least  $2n$  input lines (two lines for every pair of bits being compared). As previously discussed, at least two output lines are necessary. Based on the proof presented in this section, exactly  $2^n$  product term lines are necessary for any two-level NOR-NOR implementation. Hence, the design based on Equation (2), which was discussed in the previous section, is optimal. In the next three sections a PLA implementation of a self-testing comparator based on this design is analyzed in detail.

## V. FAULT-FREE OPERATION OF THE COMPARATOR

In the previous section we proved that any self-testing comparator implemented as a single two-level NOR-NOR PLA must have  $2^n$  product terms. In this section and the two subsequent sections we discuss a specific self-testing comparator based on Equation (2) in Section III which satisfies this necessary property.

Although a comparator based on Equation (2) has been discussed in the literature, we could find no rigorous proof that it indeed functions as a comparator. For completeness, we present such a proof in this section. To prove that, with no faults, the circuit described by Equation (2) is a comparator, we first show that if  $A = B$ , the output is (0,1) or (1,0). Then we show that if  $A \neq B$ , the output is (0,0) or (1,1).

If  $A = B$ , there are exactly  $n$  ones and  $n$  zeros at the inputs. If  $U$  is a set of integers  $U = \{i \mid a_i = 0\}$ , then for every integer  $j$  such that  $j \in (I_n - U)$ ,  $a_j = b_j = 1$ . Thus,  $b'_j = 0$ , and the one product term that corresponds to  $Q = U$  in Equation (2) is

selected. Every other product term includes the literal  $a_j$  for some  $j \in (I_n - U)$  or  $b'_i$  for some  $i \in U$ . Hence, all of the other product terms are set to 0. Thus, only the one output line connected to the single selected product term is set to 0, and the output is (0,1) or (1,0).

If  $A \neq B$ , the two bit-vectors differ by at least one bit. Consider the product term

$$NOR\{\{a_i | i \in Q\} \cup \{b'_j | j \in (I_n - Q)\}\} \quad (3)$$

for some  $Q \subset I_n$ . Assume that  $A$  and  $B$  differ in bit  $\tau$ ,  $\tau \in I_n$ , so  $a_\tau = 1$  and  $b'_\tau = 1$  in the input  $AB$ . If  $\tau \in Q$ , the product term is set to 0 since it contains the literal  $a_\tau$ . If  $\tau \notin Q$ , the product term is set to 0 since it contains the literal  $b'_\tau$ . Hence, all of the product terms are set to 0 and the output is (1,1).

Since the two bit-vectors differ by at least one bit, if there does not exist any integer  $\tau \in I_n$  such that  $a_\tau = 1$  and  $b'_\tau = 1$ , there must exist an integer  $s \in I_n$  such that  $a_s = 0$  and  $b'_s = 0$  in the input  $AB$ . If  $AB$  doesn't select any product term, the output is (1,1). Assume that the product term that corresponds to  $Q = Q_1$  (Equation (3)) is selected. If  $s \in Q_1$ , consider the set  $Q_2 = Q_1 - \{s\}$ . Since  $Q_2 \subset Q_1$ ,  $I_n - Q_2 = I_n - Q_1 + \{s\}$ , and  $b'_s = 0$ , the product term that corresponds to  $Q = Q_2$  will also be selected. If  $s \notin Q_1$ , consider the set  $Q_3 = Q_1 + \{s\}$ . Since  $a_s = 0$  and  $I_n - Q_3 \subset I_n - Q_1$ , the product term that corresponds to  $Q = Q_3$  will also be selected. Thus, either the product terms corresponding to  $Q_1$  and  $Q_2$  will be selected, or the product terms corresponding to  $Q_1$  and  $Q_3$  will be selected. The number of elements in  $Q_1$  is one greater than the number of elements in  $Q_2$  and is one less than the number of elements in  $Q_3$ . Hence, either  $|Q_2|$  and  $|Q_3|$  are even while  $|Q_1|$  is odd, or  $|Q_2|$  and  $|Q_3|$  are odd while  $|Q_1|$  is even. Thus, the product terms corresponding to  $Q_2$  and  $Q_3$  are connected to the same output line, which is different from the output line to which the product term corresponding to  $Q_1$  is connected. Therefore, product terms connected to both output lines are always selected and the output is (0,0).

## VI. IDENTIFICATION AND ELIMINATION OF UNDETECTABLE FAULTS

Given that the circuit described by Equation (2) functions as claimed when it is *fault-free*, it remains to be shown that the circuit is self-testing with respect to any single fault in the fault model described in Section II. Specifically, it must be shown that for any such fault there exists a code input that results in a noncode output (0,0) or



(1,1) from the comparator. In this section it is shown that there are a few faults in the fault model with respect to which the circuit is *not* self-testing. We refer to these problematic faults as *undetectable* faults. Layout guidelines that prevent these faults from occurring in the actual circuit are discussed.

#### *A. A Short Between Adjacent Product Term Lines*

One of the possible faults is a short between two adjacent product term lines that forces both of the lines to logic 1 when they are supposed to be carrying different values (fault type (D)). If the two product term lines are connected to the same output line, there is no code input that results in a noncode output. In fact, the circuit continues to function correctly despite this fault. The reason for this is that if one of the product term lines connected to an output line is selected, that output line is set to logic 0 regardless of the value of any other product term connected to it. It is undesirable to allow this fault to remain undetected since the situation may deteriorate in time and intermittently cause weak 0 or weak 1 faults that will not be detected and will later combine with an additional fault to cause more serious undetectable faults.

As indicated by Wang and Avizienis[29], the possibility that this undetectable fault will occur can be eliminated by ensuring that product term lines connected to the same output line are not adjacent. Since the same number of product term lines are connected to each output line, this guideline is easy to obey and incurs no penalty in terms of the size or performance of the circuit. The guideline is satisfied by simply alternating between product term lines connected to one output line and those connected to the other line.

#### *B. A Short Between a Product Term Line and an Output Line*

Another potentially undetectable fault is a short between a product term line,  $P_i$ , and an output line,  $c_m$ , where there is no device at the crosspoint of the two lines. This fault is undetectable if whenever the two lines are supposed to carry a different logic value, they are both forced to logic 1.

The short between  $P_i$  and  $c_m$  is not detectable since the faulty circuit will behave as follows: For the code input  $XX$  that is supposed to select  $P_i$ ,  $P_i$  is supposed to be at logic 1 and  $c_m$  at logic 1 (since the other output line,  $c_m$ , is supposed to be at logic 0). Hence there is no change in the output from the circuit. On the other hand,  $P_i$

is supposed to be at logic 0 and  $c_m$  is supposed to be at logic 1 for every code input,  $YY$ , such that  $YY \neq XX$  and the number of  $a_i$  ( $i \in L_n$ ) inputs that are at logic 0 in  $YY$  has the same parity as the number of  $a_i$  inputs that are at logic 0 in  $XX$ . For these code inputs,  $P_i$  is forced to logic 1 but this has no effect on  $c_m$  which is supposed to be at logic 0. For the remaining  $2^{n-1}$  code inputs,  $P_i$  is supposed to be at logic 0 and  $c_m$  is supposed to be as logic 0. Hence there is no change in the output from the circuit.

The short between  $P_i$  and  $c_m$  can be made detectable if it is possible to ensure that when  $P_i$  is at logic 0, it forces  $c_m$  to logic 0 as well. In NMOS, this can be done by using large crosspoint devices in the AND array so that a single device can pull down two load devices — the output line pull-up as well as the product term line pull-up. In CMOS, this can be done by using large crosspoint devices in the AND array so that a single device can discharge the precharged output line and product term line together within the circuit's clock period. Unfortunately, larger AND array crosspoint devices lead to a larger PLA that is also slower due to larger capacitances.

It is possible that due to a short between a product term line,  $P_i$ , and an output line,  $c_m$ , both lines always assume the value at which  $c_m$  is supposed to be. Using arguments similar to the above, it can be shown that, in this case, the short is undetectable regardless of whether or not there is a device at the crosspoint of the two lines[25]. This short can also be made detectable by using large AND array crosspoint devices that ensure that when  $P_i$  is at logic 0, it forces  $c_m$  to logic 0 as well.

### C. Shorts Resulting in Simultaneous Weak 0 and Weak 1 Faults

In this subsection we consider the possibility that, due to a short, two lines that are supposed to carry complementary values are both forced to a value between logic 0 and logic 1. The result is a weak 0 fault on one of the lines and a weak 1 fault on the other line. Such shorts may be undetectable by any code input. To show that the circuit is not self-testing with respect to such a short, it is sufficient to show that the fault is undetectable under the worst possible combination of devices that misinterpret the values on the lines.

1) *A Short Between Adjacent Product Term Lines:* As discussed in Subsection A, adjacent product term lines should be connected to different output lines. If a short between two product term lines,  $P_i$  and  $P_j$ , forces both to a value between logic 0 and



logic 1 when they are supposed to be carrying different values, this fault may be undetectable. For a code input  $XX$ , the short can affect the output only if  $XX$  is supposed to select either  $P_i$  or  $P_j$ . Without loss of generality, assume that  $XX$  is supposed to select  $P_i$ . All other product term lines (including  $P_j$ ) are not supposed to be selected by  $XX$ . However, a short between  $P_i$  and  $P_j$  can cause the OR array device connected to  $P_i$  to misinterpret is as logic 0 and the device connected to  $P_j$  to misinterpret is as logic 1. Hence, despite the fault, only one product term line ( $P_j$ ) is interpreted as being selected and the output from the circuit is a code output. Thus, this short is not detected by any code input.

It can be shown that there exists a *noncode* input that, due to the short between product term lines, results in code output. Hence this short, that is not detectable by code inputs, can mask noncode inputs. Thus, the PLA should be laid out in such a way that either this short cannot occur, or if it does occur, both lines are guaranteed to be forced to the same logic value instead of some value between logic 0 and logic 1.

We have already shown that the crosspoint devices in the AND array should be made large enough so that they can pull down both the product term line and an output line that it may be shorted to. Assuming that the same pull-ups are used for the product term lines and the output lines, each crosspoint device in the AND array is also able to pull down *two* product term lines. Hence, a short between two product term lines is guaranteed to force them both to logic 0 when they are supposed to be carrying complementary values. It will be shown in Section VII that this ensures that the short can be detected by some code input.

2) *A Short Between Adjacent Input Lines:* A short between adjacent input lines may also be undetectable by any code input if, whenever the lines are supposed to be carrying complementary values, both lines are forced to a value between logic 0 and logic 1. Consider a short between two adjacent input lines  $a_h$  and  $a_j$  ( $h \neq j$ ). There exists a code input  $XX = (x_{n-1}, \dots, x_0, x'_{n-1}, \dots, x'_0)$  for which  $a_h$  is supposed to be at logic 0 and  $a_j$  is supposed to be at logic 1. Clearly  $x_h = 0$  and  $x_j = 1$  so

$$XX = (x_{n-1}, \dots, x_{h+1}, 0, x_{h-1}, \dots, x_{j+1}, 1, x_{j-1}, \dots, x_0, \\ x'_{n-1}, \dots, x'_{h+1}, 1, x'_{h-1}, \dots, x'_{j+1}, 0, x'_{j-1}, \dots, x'_0)$$

Assume that the single product term line that is supposed to be selected by  $XX$  is  $P_i$ . Since  $x_h = 0$ , there is a device  $CA_{hi}$  at the crosspoint of the input line  $a_h$  and the product term line  $P_i$ . Assume that, due to the short, the value of both  $a_h$  and  $a_j$  is

forced to some value between logic 0 and logic 1 and that  $CA_{hi}$  misinterprets line  $a_h$  to be at logic 1. Hence product term line  $P_i$  is *not* selected by code input  $XX$ . In the fault-free circuit, the code input

$$YY = (x_{n-1}, \dots, x_{h+1}, 0, x_{h-1}, \dots, x_{j+1}, 0, x_{j-1}, \dots, x_0, \\ x'_{n-1}, \dots, x'_{h+1}, 1, x'_{h-1}, \dots, x'_{j+1}, 1, x'_{j-1}, \dots, x'_0)$$

is supposed to select product term line  $P_k$ . Hence there is a device  $CA_{jk}$  at the crosspoint of input line  $a_j$  and product term line  $P_k$ . Assume that, due to the short, when the input is  $XX$ ,  $CA_{jk}$  misinterprets  $a_j$  to be a logic 0 although it is supposed to be at logic 1. In addition, we assume that  $CA_{hi}$  and  $CA_{jk}$  are the only AND array crosspoint devices that misinterpret the values of  $a_h$  and  $a_j$  (in particular  $CA_{hi}$  interprets  $a_h$  *correctly*). Under these assumptions, all the input lines that are supposed to be at logic 0 when the input is  $YY$  are interpreted as being at logic 0 by all the AND array crosspoint devices connected to  $P_k$  when the input is  $XX$ . Hence  $P_k$  is selected by input  $XX$  while  $P_i$  is not selected by  $XX$ . Since no other crosspoint devices are effected by the short, no other product term line except  $P_k$  is selected by  $XX$ , and the output is a code output. This short does not affect the output from the circuit for any other code input since such input selects a product term other than  $P_k$  or  $P_i$ . Hence, the short is not detectable by any code input.

In the fault-free circuit, the *noncode* input

$$W = (x_{n-1}, \dots, x_{h+1}, 0, x_{h-1}, \dots, x_{j+1}, 1, x_{j-1}, \dots, x_0, \\ x'_{n-1}, \dots, x'_{h+1}, 1, x'_{h-1}, \dots, x'_{j+1}, 1, x'_{j-1}, \dots, x'_0)$$

does not select *any* product term and the output is noncode. However, due to the short described above between  $a_h$  and  $a_j$ ,  $W$  selects  $P_k$  and the result is a *code* output from the circuit. Hence this short, that is not detectable by code inputs, masks a noncode input.

It can be shown that if the adjacent input lines are  $a_h$  and  $b'_j$  a short between these lines may also be undetectable by code inputs and can mask noncode inputs [25]. Thus, in order to ensure that the comparator is self-testing, it is necessary to prevent shorts between input lines that can force both lines to a value between logic 0 and logic 1 from occurring. This can be done by laying out the PLA so that the separation between input lines is large enough that the probability of a short between them is negligible. Alternatively, the circuits that drive the inputs of the PLA can be designed so that a single pull-down device can overcome *two* pull-up devices so that a short between input

lines when they are supposed to be carrying different values will always results in both being forced to logic 0. Unfortunately, these solutions lead to a larger PLA that is also slower due to larger capacitances.

3) *A Short Between an Input Line and a Product Term Line:* Using arguments similar to the above, it can be shown that if a short between an input line and a product term line is allowed to force both of them to a value between logic 0 and logic 1, an undetectable fault, that can mask noncode inputs, may result. Here again, one way to prevent this situation is to guarantee that when the lines are supposed to be at complementary values they are both always forced to logic 0. This can be done using large pull-down devices in the circuits that drive the inputs of the PLA and using large AND array crosspoint devices. A single AND array crosspoint device or a single pull-down in an input driver must be able to overcome both the pull-up device of the input driver and the pull-up device of the product term line.

#### *D. Layout Guidelines for Eliminating Undetectable Faults*

In the previous three subsections we identified several possible faults that are not detectable by any code inputs. All of these faults are shorts between adjacent or crossing lines. In particular, any short that results in both lines being forced to a value between logic 0 and logic 1 when they are supposed to be carrying complementary values may lead to an undetectable fault. The layout guidelines for preventing these faults from occurring in the actual circuit are summarized below.

- (1) Adjacent product term lines must be connected to OR array crosspoint devices that control different output lines.
- (2) The AND array crosspoint devices must be large enough so that a single device can pull down two pull-ups — a product term line pull-up and an output line pull-up or two product term line pull-ups.
- (3) The circuits that drive the inputs of the PLA must be designed so that a single pull-down device can overcome *two* pull-up devices.
- (4) The separation between adjacent input lines and between adjacent product term lines should be larger than the minimum separation required by the design rules. This can help reduce the probability of a short between adjacent lines.

## VII. THE SELF-TESTING PROPERTY OF THE COMPARATOR

In the previous section it was shown that the proposed comparator is *not* self-testing with respect to some of the possible faults, unless certain guidelines about the layout of the circuit and the size of some of its devices are followed. In this section it is shown that the circuit is self-testing with respect to all the other faults in the fault model. It is assumed that some measures, such as those discussed in the previous section, are taken so that the undetectable faults cannot occur. In particular, it is assumed that if there is a short between two lines and the lines are supposed to be carrying complementary values, the value of one of the lines is modified so that they both carry the same logic value.

### A. A Weak 0 and/or Weak 1 Fault on a Single Input Line

1) *A Weak 0 Fault:* Assume that the input line with a weak 0 fault is  $a_k$  for some  $k \in I_n$ . By definition, there is at least one AND array crosspoint device,  $CA_{ki}$ , connected to  $a_k$  that always misinterprets a logic 0 on  $a_k$  as a logic 1. Hence, the device  $CA_{ki}$  is always turned on. Thus, the product term line  $P_i$  that is connected to  $CA_{ki}$  can never be selected. Therefore the code input that is supposed to select  $P_i$  results in no product term line being selected and the output is noncode (1,1). An identical argument can be made regarding a weak 0 fault on a  $b'_j$  ( $j \in I_n$ ) input line.

In the presence of a weak 0 fault on one of the input lines, for every crosspoint device which misinterprets the input line to be a logic 1 when it is supposed to be a logic 0, the code input that selects the corresponding product term line in the fault-free circuit results in a (1,1) output. Thus the number of code inputs that detect this fault varies between 1 and  $2^{n-1}$ , depending on the number of affected crosspoint devices.

2) *A Weak 1 Fault:* Assume that the line with a weak 1 fault is  $a_k$  for some  $k \in I_n$ . By definition, there is at least one AND array crosspoint device,  $CA_{ki}$ , connected to  $a_k$  that always misinterprets a logic 1 on  $a_k$  as a logic 0. We denote the product term line connected to that crosspoint device by  $P_i$ . In the fault-free circuit,  $P_i$  is selected by some code input  $XX = (x_{n-1}, \dots, x_0, x'_{n-1}, \dots, x'_0)$ . Since there is a device at the crosspoint of  $a_k$  and  $P_i$ , the literal  $a_k$  is in the product term that corresponds to  $P_i$ . Hence  $x_k = 0$ . Thus,

$$XX = (x_{n-1}, \dots, x_{k+1}, 0, x_{k-1}, \dots, x_0, x'_{n-1}, \dots, x'_{k+1}, 1, x'_{k-1}, \dots, x'_0).$$

In the fault-free circuit, the code input

$$YY = (x_{n-1}, \dots, x_{k+1}, 1, x_{k-1}, \dots, x_0, x'_{n-1}, \dots, x'_{k+1}, 0, x'_{k-1}, \dots, x'_0)$$

selects some other product term line  $P_j$ . Since  $CA_{k1}$  misinterprets a logic 1 on  $a_k$  to be a logic 0, code input  $YY$  selects  $P_i$ . Since there is no device at the crosspoint of  $a_k$  and  $P_j$ ,  $P_j$  is independent of  $a_k$ . Thus  $YY$  also selects  $P_j$  despite the fault.

Since the number of  $a_i$  ( $i \in I_n$ ) inputs that are at logic 0 in  $XX$  has a different parity from the number of  $a_i$  inputs that are at logic 0 in  $YY$ ,  $P_i$  and  $P_j$  are connected to different output lines (see Equation (2) Section III). Since in the faulty circuit, the code word  $YY$  selects both  $P_i$  and  $P_j$  the output is (0,0). An identical argument can be made regarding a weak 1 fault on a  $b'_j$  ( $j \in I_n$ ) input line.

#### B. A Short Between Two Adjacent Input Lines

As previously mentioned, we assume that appropriate layout guidelines are followed so that a short between lines always forces both of the lines to the same logic value rather than to a value between logic 0 and logic 1. Since the inputs to the comparator are all the bits from one of the functional modules and the complements of all the bits from the other module, no two input lines are supposed to have the same value for all code inputs. If the two adjacent shorted lines are  $a_k$  and  $b'_k$  ( $0 \leq k \leq n-1$ ), every code input is transformed to noncode input which, as previously shown, results in (0,0) or (1,1) output. Any other two input lines are supposed to transfer different values for half of the code inputs. For these code inputs, the short forces a change in value on one of the lines. Since we assume that there are no other faults, this is equivalent to noncode input which, as previously shown, results in (0,0) or (1,1) output.

#### C. A Weak 0 or Weak 1 Fault on a Single Product Term Line

Each product term line is connected to only one output line. Hence, a weak 0 fault on a product term line is simply a stuck-at-1 fault and a weak 1 fault is a stuck-at-0 fault.

1) *A Weak 1 (s-a-0) Fault:* If one of the product terms is s-a-0, for the code input that is supposed to select that product term, all product terms are set to 0 and the output is (1,1).

2) *A Weak 0 (s-a-1) Fault:* Assume that the product term line  $P_i$  that corresponds to set  $Q = Q_1$  in Equation (3), is s-a-1. For any code input that selects a product term corresponding to some  $Q = Q_2 \in I_n$ , where the parity of  $|Q_1|$  and  $|Q_2|$  are different,

product terms connected to both output lines are selected, and the output is (0,0). Thus half the code inputs will result in a (0,0) output due to the s-a-1 fault on  $P_i$ .

#### *D. A Short Between Two Adjacent Product Term Lines*

Since only one product term line is supposed to be selected by every code input, for any pair of adjacent product term lines,  $P_i$  and  $P_j$ , there is one code input that is supposed to select  $P_i$  but not  $P_j$  and there is another code input that is supposed to select  $P_j$  but not  $P_i$ . We consider the three possible effects of the short when the lines are supposed to carry complementary values:

(1) Both lines are always forced to logic 0: In this case, for the two code inputs that correspond to the two product terms (i.e. that are supposed to select them), no product term line will be set to 1 and the output will be (1,1).

(2) Both lines are always forced to logic 1: Since the two product term lines are connected to different output lines, for the two code inputs that correspond to these product term lines, the output will be (0,0).

(3) Both product term lines always assume the value of one of the lines: Assume that the two lines are  $P_i$  and  $P_j$ , and that  $P_i$  always dominates. The code input  $YY$ , that is supposed to select  $P_j$ , does not select it, since  $P_j$  is pulled to logic 0 by  $P_i$ , which is not selected by  $YY$ . Hence  $YY$  does not select any product term and the output is (1,1). The code input  $XX$ , that selects  $P_i$  also selects  $P_j$  which is pulled to logic 1 by  $P_i$ . Since adjacent product term lines are connected to different output lines,  $XX$  results in (0,0) output.

#### *E. A Weak 0 or Weak 1 Fault on a Single Output Line*

The output lines do not fan out within the comparator circuit. Thus, we need only consider the value on the output line at the point of interface with the "outside world." Hence, a weak 0 fault on a product term line is equivalent to a stuck-at-1 fault and a weak 1 fault is equivalent to a stuck-at-0 fault.

Based on Equation (2), any code input where the number of  $a_i$  ( $i \in I_n$ ) bits that are at logic 0 is odd, is supposed to result in the output  $(c_1, c_0) = (1, 0)$ . Thus, in the faulty circuit, if  $c_0$  is s-a-1, the output is (1,1), and if  $c_1$  is s-a-0, the output is (0,0). A similar argument can be made for any code input where the number of  $a_i$  bits that are at logic 0 is even and the output is supposed to be  $(c_1, c_0) = (0, 1)$ . Hence  $2^{n-1}$  code inputs



will detect a s-a-1 on  $c_0$  and a s-a-0 on  $c_1$  while the other  $2^{n-1}$  code inputs will detect a s-a-0 on  $c_0$  and a s-a-1 on  $c_1$ .

#### *F. A Short Between Two Adjacent Output Lines*

There are only two output lines that are supposed to carry different values for every code input. Hence, a short will result in (0,0) or (1,1) output for every code input.

#### *G. A Short Between an Input Line and a Crossing Product Term Line*

Assume that the short is between input line  $a_k$  and product term line  $P_i$ . Let  $XX$  denote the code input that selects  $P_i$  in the fault-free circuit. We must consider the case where  $a_k$  is connected to a crosspoint device that is connected to  $P_i$  ( $CA_{ki}$  exists) as well as the case where  $CA_{ki}$  does not exist.

If  $CA_{ki}$  exists, every one of the  $2^{n-1}$  code inputs for which  $a_k$  is supposed to be at logic 1, is supposed to result in  $P_i$  at logic 0. The code input  $XX$  is the only code input for which  $a_k$  is supposed to be at logic 0 while  $P_i$  is supposed to be at logic 1. For the rest of the  $2^{n-1}-1$  code inputs, both  $a_k$  and  $P_i$  are supposed to be at logic 0.

If  $CA_{ki}$  does not exist, the product term corresponding to  $P_i$  includes the literal  $b'_k$ . For the  $2^{n-1}$  code inputs with  $b'_k$  at logic 1, both  $a_k$  and  $P_i$  are supposed to be at logic 0. For the code input  $XX$ ,  $b'_k$  is supposed to be at logic 0, and both  $a_k$  and  $P_i$  are supposed to be at logic 1. For the rest of the  $2^{n-1}-1$  code inputs,  $b'_k$  is supposed to be at logic 0,  $a_k$  is supposed to be at logic 1, and  $P_i$  is supposed to be at logic 0. Thus, if  $CA_{ki}$  does not exist, there is no code input for which  $a_k$  is supposed to be at logic 0 and  $P_i$  is supposed to be at logic 1.

As in the proof for a short between product term lines, we consider the three possible effects of the short when  $a_k$  and  $P_i$  are supposed to carry complementary values:

(1) Both lines are forced to logic 0: If  $CA_{ki}$  exists, for the code input  $XX$ ,  $P_i$  is supposed to be the only selected product term line, while  $a_k$  is supposed to be at logic 0. We assume that, due to the short,  $P_i$  is forced to logic 0 by  $a_k$ . Hence, no product term line is selected and the output is (1,1).

On the other hand, if  $CA_{ki}$  does not exist, the literal  $a_k$  is not included in the product term that corresponds to  $P_i$ . Hence, the code input that selects  $P_i$  in the fault-free circuit is of the form:



$$XX = (x_{n-1}, \dots, x_{k+1}, 1, x_{k-1}, \dots, x_0, x'_{n-1}, \dots, x'_{k+1}, 0, x'_{k-1}, \dots, x'_0).$$

Let  $YY$  be one of the  $2^{n-1}-1$  code inputs such that  $YY \neq XX$  and  $YY$  is also of the form

$$YY = (y_{n-1}, \dots, y_{k+1}, 1, y_{k-1}, \dots, y_0, y'_{n-1}, \dots, y'_{k+1}, 0, y'_{k-1}, \dots, y'_0).$$

In the fault-free circuit  $YY$  selects some product term line  $P_j$ . Since there is no device at the crosspoint of  $a_k$  and  $P_j$ ,  $P_j$  is independent of  $a_k$  so the short between  $a_k$  and  $P_i$  cannot affect  $P_j$ . Thus  $P_j$  is selected by  $YY$  despite the fault. In the fault-free circuit, the code input

$$ZZ = (y_{n-1}, \dots, y_{k+1}, 0, y_{k-1}, \dots, y_0, y'_{n-1}, \dots, y'_{k+1}, 1, y'_{k-1}, \dots, y'_0)$$

selects some product term  $P_s$ . Since there is no device at the crosspoint of  $b'_k$  and  $P_s$ ,  $P_s$  is independent of  $b'_k$ . For the code input  $YY$ ,  $a_k$  is supposed to be at logic 1 and  $P_i$  at logic 0. Due to the fault,  $P_i$  forces  $a_k$  to logic 0. Therefore,  $YY$  selects  $P_s$  as well as  $P_j$ . Since the number of  $a_i$  ( $i \in I_n$ ) inputs that are at logic 0 in  $YY$  has a different parity from the number of  $a_i$  inputs that are at logic 0 in  $ZZ$ ,  $P_j$  and  $P_s$  are connected to different output lines (see Equation (2) Section III). Hence, for the code input  $YY$  the output is (0,0).

(2) Both lines are forced to logic 1: Let  $YY$  be one of the  $2^{n-2}$  code inputs for which  $a_k$  is supposed to be at logic 1 and the number of  $a_i$  inputs that are at logic 0 in  $YY$  has a different parity from the number of  $a_i$  inputs that are at logic 0 in  $XX$ . Due to the short, when the input is  $YY$ ,  $a_k$  forces  $P_i$  (that is supposed to be at logic 0) to logic 1. In addition, as in the fault-free circuit,  $YY$  selects another product term that controls a different output line from  $P_i$ . Hence the output from the circuit is (0,0).

(3) Both lines are always forced to the value of  $a_k$  or they are always forced to value of  $P_i$ :

(a) Line  $a_k$  always dominates: The proof is identical to case (2) above.

(b) Line  $P_i$  always dominates: There are at least  $2^{n-1}-1$  code inputs of the form

$$YY = (y_{n-1}, \dots, y_{k+1}, 1, y_{k-1}, \dots, y_0, y'_{n-1}, \dots, y'_{k+1}, 0, y'_{k-1}, \dots, y'_0)$$

that do not select  $P_i$  in the fault-free circuit. In the faulty circuit, if  $P_i$  always "dominates,"  $YY$  selects two product term lines that are connected to different output lines. One is the product term line selected by  $YY$  in the fault-free circuit and the other is the product term line selected by

$$ZZ = (y_{n-1}, \dots, y_{k+1}, 0, y_{k-1}, \dots, y_0, y'_{n-1}, \dots, y'_{k+1}, 1, y'_{k-1}, \dots, y'_0)$$

in the fault-free circuit. Hence, the output is  $(0,0)$ .

#### *H. A Short Between a Product Term Line and a Crossing Output Line*

Assume that the short is between product term line  $P_i$  and output line  $c_m$ , where  $m \in \{0,1\}$ . Let  $m'$  denote 0 when  $m$  is 1 and denote 1 when  $m$  is 0. Let  $XX$  denote the code input that selects  $P_i$  in the fault-free circuit.

As in the proof for a short between product term lines, we consider the three possible effects of the short when  $P_i$  and  $c_m$  are supposed to carry complementary values:

(1) Both lines are forced to logic 0: In the fault-free circuit there are at least  $2^{n-1}-1$  code inputs that do not select  $P_i$  and for which the output is  $(c_m, c_{m'}) = (1,0)$ . For any one of these inputs, due to the short,  $P_i$  forces  $c_m$  to logic 0 and the output is  $(0,0)$ .

(2) Both lines are forced to logic 1: If there is a device at the crosspoint of  $P_i$  and  $c_m$  ( $CO_{im}$  exists), in the fault-free circuit, for the code input  $XX$  that selects  $P_i$ , the output is  $(c_m, c_{m'}) = (0,1)$ . In the faulty circuit, due to the short,  $c_m$  is forced to logic 1. Since none of the product term lines are affected, the output is  $(1,1)$ . If  $CO_{im}$  does not exist, then, as discussed in Subsection B of Section VI, the fault cannot be detected by any code input.

(3) Both lines are always forced to the value of  $P_i$  or they are always forced to value of  $c_m$ :

(a) If the value of  $P_i$  always "dominates," the proof is identical to case (1) above.

(b) If the value of  $c_m$  always "dominates," then, as discussed in Subsection C of Section VI, the fault cannot be detected by any code input.

#### *I. An Extra Crosspoint Device in the AND Array*

In the fault-free circuit, every product term line,  $P_i$ , is connected to  $n$  crosspoint devices in the AND array. For every code input,  $n$  of the input lines are at logic 0 and  $n$  are at logic 1. If, due to a fault, there are  $n+1$  crosspoint devices connected to  $P_i$ , every code input turns on at least one of these devices and sets  $P_i$  to logic 0. Thus, the single code input that selects  $P_i$  in the fault-free circuit does not select  $P_i$  in the faulty circuit. Hence, for that input, no product term line is selected, and the output is  $(1,1)$ .

### J. An Extra Crosspoint Device in the OR Array

An extra crosspoint device in the OR array means that there is one product term line,  $P_i$  that is connected to both output lines. Hence, for the single code input that selects  $P_i$ , the output is (0,0).

### K. A Break in a Product Term Line

Each product term line controls one OR array crosspoint device and is controlled by  $n$  AND array pull-down devices and one pull-up (or precharge) device. All the pull-down devices are connected to the "middle" of the line. The pull-up device and the OR array crosspoint device are either connected on opposite ends of the product term line (as shown in Fig. 1) or on the same end of the line.

If the product term line pull-up device and the OR array crosspoint device are on opposite ends of the line, any break in the product term line prevents the segment of the line connected to the OR array device from being pulled up. As a result, the product term line is either floating or stuck-at-0. If the line is floating, its value is constant and independent of the input. Hence, in any case, the product term line segment that controls the output line is either stuck-at-0 or stuck-at-1. Earlier in this section it is shown that a stuck-at fault on a product term line is detectable by some code input.

If the product term line pull-up device and the OR array crosspoint device are on the same end of the line, a break in the product term line disconnects some of the AND array crosspoint devices from the segment of the line connected to the OR array device. As a result, the product term line is selected when it is not supposed to be selected. Let  $P_i$  denote the product term line that is selected by the code input  $XX = (x_{n-1}, \dots, x_h, \dots, x_0, x'_{n-1}, \dots, x'_h, \dots, x'_0)$  in the fault-free circuit. A break in  $P_i$  disconnects some AND array crosspoint device,  $CA_{hi}$ , from the segment of  $P_i$  that controls the OR array device. Since  $CA_{hi}$  is controlled by input line  $a_h$ , in the fault-free circuit,  $P_i$  can only be selected if  $a_h = 0$ . Hence,  $x_h = 0$ . In the fault-free circuit, the code input  $YY = (x_{n-1}, \dots, x'_h, \dots, x_0, x'_{n-1}, \dots, x_h, \dots, x'_0)$  selects the product term  $P_j$  where  $P_i$  and  $P_j$  are connected to different output lines. Since the crosspoint device  $CA_{hi}$  is disconnected from  $P_i$  in the faulty circuit,  $P_i$  is not affected by  $a_h$  and the code input  $YY$  selects both  $P_i$  and  $P_j$ . Hence, the output is a (0,0).

### L. A Break in an Output Line

Each output line is controlled by  $2^n-1$  OR array pull-down devices and one pull-up (or precharge) device. All the pull-down devices are connected to the "middle" of the line. The pull-up device and the output from the circuit are either on opposite ends of the output line (as shown in Fig. 1) or on the same end of the line.

If the output line pull-up device and the circuit output are on opposite ends of the line, any break in the output line prevents the segment of the line that serves as the output from the circuit from being pulled up. As a result the output line is either floating or stuck-at-0. If the line is floating, its value is constant and independent of the input. Hence, in any case, the segment of the line that serves as the circuit output is either stuck-at-0 or stuck-at-1. Earlier in this section it is shown that a stuck-at fault on an output line is detectable by some code input.

If the output line pull-up device and the circuit output are on the same end of the line, a break in the output line disconnects some of the OR array crosspoint devices from the segment of the line that is the output from the circuit. As a result, the output line is selected when it is not supposed to be selected. Let  $c_m$  denote the output line with a break. Let  $CQ_{im}$  denote an OR array crosspoint device that is disconnected from the segment of  $c_m$  that serves as the circuit output. In the fault-free circuit, the product term line  $P_i$ , that controls  $CQ_{im}$ , is selected by the code input  $XX$ . In the faulty circuit, due to the break, the crosspoint device  $CQ_{im}$  cannot affect the output line  $c_m$ . For the code input  $XX$ ,  $P_i$  is the only selected product term. Hence  $CQ_{im}$  is the only OR array crosspoint device that is turned on. Therefore neither output line is pulled down and the circuit produces the noncode output (1,1).

## VIII. IMPLEMENTATION AND APPLICATION CONSIDERATIONS

In the previous three sections it was shown that, using a single two-level NOR-NOR PLA, it is possible to implement a comparator that is self-testing with respect to any single fault that is likely to occur in MOS VLSI circuits. This result is a necessary prerequisite for the use of duplication and matching as the basic scheme for implementing error detection. However, two main problems remain to be discussed: (1) The size of the comparator, implemented as a single PLA, grows exponentially with the number of bits in the two vectors to be compared. (2) It is necessary to ensure that all the code inputs will appear as inputs to the comparator often enough so that a complete

self-test of the comparator will be performed before there is a chance for multiple faults to occur in the system.

In Section IV it was shown that a self-testing comparator implemented as a single two-level NOR-NOR PLA, must have  $2^n$  product term lines. If the output from each one of the duplicated functional modules is, say, 16 bits, this implementation is impractical since it requires  $2^{16} = 65536$  product terms. Fortunately, efficient implementations of a self-testing two-rail code checker (comparator) for large input vectors can be achieved by using checkers for smaller input vectors as *cells* that are connected together in a tree structure (Fig. 3) [15, 28].

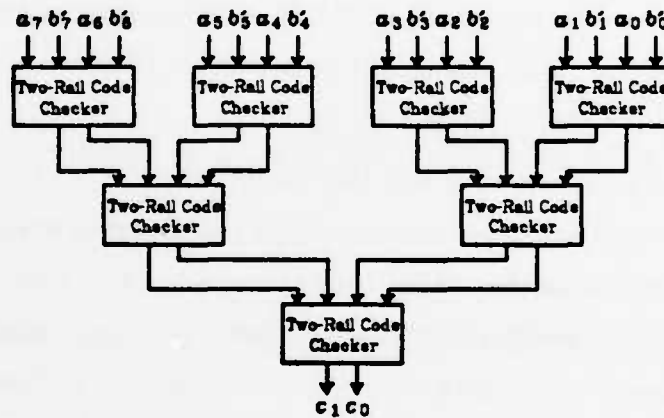


Fig. 3: A Self-Testing Two-Rail Code Checker Tree

Each cell is a self-testing comparator for relatively small bit vectors (two to six bits wide) which is implemented with a single two-level NOR-NOR PLA as outlined in Section III. A complete tree with  $h$  levels of cells, where each cell is an  $m$ -bit comparator, can be used to compare  $m^h$  bits and contains  $(m^h - 1)/(m - 1)$  cells. Hence, if the vectors to be compared are  $n$  bits wide, the number of levels in the tree is  $\lceil \log_m n \rceil$  while the total number of cells in the tree is at most  $(n - 1)/(m - 1)$ . Thus the number of cells is (approximately) linearly related to  $n$ . Hence, tree-structured cellular implementations of self-testing comparators are practical for large input bit vectors.

In the cellular tree-structured implementation of the comparator, a noncode output from any one of the cells presents a noncode input to the cells at the next level. This forces the output from the entire tree to be noncode. Hence, the tree-structured implementation preserves the self-testing property of the cells.

If duplication and matching is used for error detection, the first fault that occurs in

The comparator must be detected before additional faults can occur in the comparator or in the functional modules. Thus, a set of code words that achieves a complete self-test of the comparator must appear as inputs to the comparator within a time interval that is significantly smaller than the mean time between failures for the two functional modules and the comparator together. Based on the results of sections IV and VII, a complete self-test of a comparator implemented as a single NOR-NOR PLA requires all  $2^n$  code words to appear at the inputs. If  $n$  is large, this requirement may imply that the complete self-test takes so much time that there is an unacceptably high probability that additional faults may occur in the comparator or functional modules before the self-test is completed. Fortunately, for the tree-structured cellular implementation, the number of code inputs required for a complete-self test is only  $2^m$ , where  $m$  is the size of the bit vectors compared by each cell [5, 15]. Thus, if the cells are 2-bit comparators, four code inputs are sufficient for a complete self-test of the entire tree.

Even with the relatively small number of code inputs needed for a complete self-test, it may still be difficult to satisfy the requirement that certain code words appear as inputs to the comparator with some specified frequency, as implied by the assumptions in Section III. We assume that the system consists of subsystems that interact with each other. Each subsystem is implemented with duplicate functional modules and a self-testing comparator so that the failure of a particular subsystem is detected by the other subsystems by simply observing the output from the corresponding comparator [24]. The comparison of the outputs of the two modules that make up each subsystem is performed at the interface between the subsystem and the rest of the system.

If the "subsystems" are low-level passive circuits such as an ALU or an instruction decoder within a processor, it may not be possible to ensure that the necessary outputs will be generated by the modules. Hence, duplication and matching is *inappropriate* at this level.

Duplication and matching is an attractive scheme for implementing error detection if the subsystems are high-level, "intelligent" modules that interact with similar modules. Examples of such high-level subsystems are the computation nodes or the communication nodes in a multicomputer system [24]. In this case, the subsystem may periodically initiate action that causes it to generate all the necessary patterns at its interface with the other subsystems. The subsystem initiating the self-test of its comparator can inform the other subsystems that the next "message" is simply a test



and should not be interpreted as "real work."

### IX. SUMMARY AND CONCLUSIONS

We have presented a new fault model for MOS PLAs. This model incorporates several types of faults that are likely to occur in VLSI circuits but are not taken into account in previously published models. Using this more realistic model, it was shown that the widely accepted design of a "self-testing" comparator implemented as a NOR-NOR PLA results in a comparator that is self-testing with respect to any single fault *only* if certain guidelines regarding the physical layout of the circuit are followed. Following these guidelines requires additional silicon area and reduces performance.

The use of duplication and matching of high-level modules for error detection appears especially attractive in view of the difficulties in analyzing and verifying the self-testing properties of the comparator. Despite the simple structure of the proposed self-testing comparator, the analysis and verification of its self-testing properties are surprisingly lengthy and complex. It is therefore doubtful that the effects of faults on large VLSI circuits, such as microprocessors, can be predicted reliably. Furthermore, while restrictions on the layout of the comparator are acceptable in order to enhance its testability, such restrictions cannot be tolerated in the layout of large chips, where one of the major goals is to implement as much functionality as possible per unit area.

Since the area taken up by the comparator may be of concern, it was shown that the proposed comparator is optimal with respect to the area it occupies. Specifically, it is shown that if a single two-level NOR-NOR PLA is used to implement a self-testing comparator of two  $n$ -bit vectors, an optimal design must include  $2n$  input lines,  $2^n$  product term lines and 2 output lines. Furthermore,  $2^n$  code inputs are necessary for a complete self-test of any such circuit.

The effectiveness of self-testing comparators as critical elements in duplication and matching schemes for error detection is dependent on the system within which they are employed. If the duplicated functional modules are simple, low-level, passive circuits, it may not be possible to ensure that the comparator will go through a complete self-test often enough and the scheme may eventually fail due to an undetected fault in the comparator. However, if the duplicated modules are high-level subsystems, the use of self-testing comparators in a duplication and matching scheme is an effective way to implement error detection in VLSI systems.



## Acknowledgements

We would like to thank Richard Fujimoto and Clark Thompson for reviewing an early draft of this paper.

## References

1. D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers* C-22(3) pp. 263-269 (March 1973).
2. T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 29-33 (June 1982).
3. A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers* C-20(11) pp. 1322-1330 (November 1971).
4. A. Avizienis, "Design Diversity - The Challenge of the Eighties," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 44-45 (June 1982).
5. D. C. Bossen, D. L. Ostapko, and A. M. Patel, "Optimum Test Patterns for Parity Networks," *AFIPS Proceedings* 37 pp. 63-68 (1970).
6. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *Proceedings of the IFIPS*, Edinburgh, Scotland, pp. 878-883 (August 1968).
7. X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* C-31(7) pp. 658-671 (July 1982).
8. M. L. Ciacelli, "Fault Handling on the IBM 4341 Processor," *11th Fault-Tolerant Computing Symposium*, Portland, Main, pp. 9-12 (June 1981).
9. B. Courtois, "Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits," pp. 341-350 in *VLSI 81*, ed. J. P. Gray, Academic Press (1981).
10. R. P. Davidson, M. L. Harrison, and R. L. Wadsack, "BELLMAC-32: A Testable 32 Bit Microprocessor," *1981 International Test Conference Proceedings*, Philadelphia, PA, pp. 15-20 (October 1981).
11. E. A. Doyle, "How Parts Fail," *IEEE Spectrum* 18(10) pp. 36-43 (October 1981).
12. A. D. Friedman and P. R. Memon, *Fault Detection in Digital Circuits*, Prentice Hall (1971).
13. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers* C-29(6) pp. 527-531 (June 1980).
14. R. T. Howard, "Packaging Reliability: How to Define and Measure It," *32nd Electronic Components Conference*, San Diego, CA, pp. 376-384 (May 1982).
15. J. Khakbaz and E. J. McCluskey, "Concurrent Error Detection and Testing for Large PLA's," *IEEE Journal of Solid-State Circuits* SC-17(2) pp. 386-394 (April 1982).
16. G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 303-310 (June 1982).
17. M. A. Marouf and A. D. Friedman, "Efficient Design of Self-Checking Checker for any

- m-Out-of-n Code," *IEEE Transactions on Computers* C-27(6) pp. 482-490 (June 1978).
18. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).
  19. D. L. Ostapko and S. J. Hong, "Fault Analysis and Test Generation for Programmable Logic Arrays," *IEEE Transactions on Computers* C-28(9) pp. 617-627 (September 1979).
  20. R. A. Rasmussen, "Automated Testing of LSI," *Computer* 15(3) pp. 69-78 (March 1982).
  21. C. H. Séquin, "Managing VLSI Complexity: An Outlook," *IEEE Proceedings* 71(1) pp. 149-166 (January 1983).
  22. D. P. Siewiorek and D. Johnson, "Design Methodology for High Reliability Systems: The Intel 432," pp. 621-636 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).
  23. K. Son and D. K. Pradhan, "Completely Self-Checking Checkers in PLAs," *1981 International Test Conference*, Philadelphia, PA, pp. 231-237 (October 1981).
  24. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *IEEE International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).
  25. Y. Tamir, "Fault Tolerance for VLSI Multicomputers," Ph.D. Dissertation (in preparation).
  26. M. M. Tsao, A. W. Wilson, R. C. McGarity, C. Tseng, and D. P. Siewiorek, "The Design of C.fast: A Single Chip Fault Tolerant Microprocessor," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 63-69 (June 1982).
  27. R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *The Bell System Technical Journal* 57(5) pp. 1449-1474 (May-June 1978).
  28. J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland (1978).
  29. S. L. Wang and A. Avizienis, "The Design of Totally Self Checking Circuits Using Programmable Logic Arrays," *9th Fault-Tolerant Computing Symposium*, Madison, WI, pp. 173-180 (June 1979).

PUBLICATIONS ON COMPUTER AIDS FOR DESIGN AND LAYOUT

## COMPUTER AIDS FOR DESIGN AND LAYOUT

The following section contains papers and reports relating to research in Computer Aids For Design and Layout. They describe work which was wholly or in part performed under the sponsorship of the DARPA grant.

- (1) J.K. Ousterhout, "Switch-level Delay Models for MOS VLSI", to appear, *21st Design Automation Conference*, June 1984.
- (2) J.K. Ousterhout, G.T. Hamachi, R.N. Mayo, W.S. Scott, and G.S. Taylor, "A Collection of Papers on Magic," Technical Report No. UCB/CSD 83/154, Computer Science Division, University of California, Berkeley, December 1983.
- (3) G.S. Taylor and J.K. Ousterhout, "Magic's Incremental Design-Rule Checker," to appear, *21st Design Automation Conference*, June 1984.
- (4) W.S. Scott and J.K. Ousterhout, "Plowing: Interactive Stretching and Compaction in Magic," to appear, *21st Design Automation Conference*, June 1984.
- (5) G. De Micheli, A. Sangiovanni-Vincentelli and T. Villa, "Computer-aided Synthesis of Finite-State Machines," *Proc. of Int. Conf. on CAD 1983*, Santa Clara, California, Sept. 1983.
- (6) A.R. Newton and A. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation", *IEEE Trans. on Elec. Dev.*, Sept. 1983 *SIAM Journ. on Scientific and Statistical Computing*, Sept. 1983.
- (7) R. Saleh, "Iterated Timing Analysis and SPLICE1," ERL Memorandum Number UCB/ERL M84/2, Electronics Research Laboratory, University of California, Berkeley, 4 January 1984.
- (8) H. Ko and A. Sangiovanni-Vincentelli, "BLOSSOM: an Algorithm and Architecture for the Solution of Large-Scale Linear Systems" *Proceedings of the International Conference on Computer Design*, New York, Oct. 1983.
- (9) J.T. Deutsch and A.R. Newton, "A Multiprocessor-Based Implementation of Relaxation-Based Circuit Simulation," to appear *Proc. 21st Design Automation Conference*, Albuquerque, New Mexico, June 1984.
- (10) C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf: A Placement and Routing Package," to appear, *Proc. of Cust. Int. Circ. Conf.* Rochester, May 1984.
- (11) J.K. Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-3, No. 1, Jan. 1984, pp 87-100.

# Switch-Level Delay Models for Digital MOS VLSI

John K. Ousterhout

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California  
Berkeley, CA 94720  
415-642-0865

## Abstract

This paper presents fast, simple, and relatively accurate delay models for large digital MOS circuits. Delay modelling is organized around chains of switches and nodes called *stages*, instead of logic gates. The use of stages permits both logic gates and pass transistor arrays to be handled in a uniform fashion. Three delay models are presented, ranging from an RC model that typically errs by 25% to a slope-based model whose delay estimates are typically within 10% of SPICE's estimates. The slope model is parameterized in terms of the ratio between the slopes of a stage's input and output waveforms. All the models have been implemented in the Crystal timing analyzer. They are evaluated by comparing their delay estimates to SPICE, using a dozen critical paths from two VLSI designs.

**Keywords and Phrases:** timing analysis, delay, transistor models.

\*\* DRAFT \*\*    Do not    distribute    widely

# Switch-Level Delay Models for Digital MOS VLSI

## Abstract

This paper presents fast, simple, and relatively accurate delay models for large digital MOS circuits. Delay modelling is organized around chains of switches and nodes called *stages*, instead of logic gates. The use of stages permits both logic gates and pass transistor arrays to be handled in a uniform fashion. Three delay models are presented, ranging from an RC model that typically errs by 25% to a slope-based model whose delay estimates are typically within 10% of SPICE's estimates. The slope model is parameterized in terms of the ratio between the slopes of a stage's input and output waveforms. All the models have been implemented in the Crystal timing analyzer. They are evaluated by comparing their delay estimates to SPICE, using a dozen critical paths from two VLSI designs.

**Keywords and Phrases:** timing analysis, delay, transistor models.

## 1. Introduction

Over the last twenty years, a great deal of effort has been spent in the development of transistor models for use in simulation programs. The primary concern in those models has been *accuracy*: the ability to simulate exactly the real-world behavior of devices. The models have achieved high accuracy by modelling circuits with systems of differential equations. The success of the models can be seen in the popularity of circuit simulation programs such as SPICE [3].

Unfortunately, the accuracy of the circuit models comes at a high price in execution time: circuit simulators typically require several seconds of CPU time per transistor. As a result, the programs are impractical for today's state-of-the-art VLSI circuits, which contain tens or hundreds of thousands of transistors. Although there have been recent improvements in the speed of circuit simulators [2], they still require too much time for VLSI circuits.

This paper describes a different approach to transistor modelling, where speed is the primary consideration. The models treat each transistor as a perfect switch in series with a resistor. Instead of solving differential equations, the switch-level models use tables to compute the value of the series resistance. The switch-level approach results in four orders of magnitude improvement in speed: only a few hundred microseconds of execution time are needed per transistor. In spite of their simplicity, the switch-level models provide delay



estimates for digital MOS circuits that are typically within 10% of what SPICE would estimate for the same circuits.

The switch-level models achieve their accuracy and speed by capitalizing on the uniform design style used in large circuits. For example, digital VLSI circuits tend to have only a few different sizes of transistor and a few pullup-pulldown ratios, used over and over. Since the pieces of the circuit have about the same structure, they also have about the same delay properties. The delay properties of the basic constructs can be measured by running SPICE on small examples and distilling the results down to a few tables. When analyzing large circuits, delay estimates are computed quickly using the tables. If there isn't much variation in the structures used in the circuit, small tables will produce accurate results. The approximate models tend not to work as well for sensitive analog components or circuits with large variation in design style.

This paper describes and evaluates three switch-level delay models that have been implemented in Crystal, a program that locates critical timing paths in VLSI circuits [5]. Crystal's models contain two features that permit fast and accurate delay estimates. First, circuits are decomposed into chains of transistors called *stages*. Each stage is independent for purposes of delay calculation. The stage decomposition permits Crystal to handle both logic gates and pass transistors in a uniform fashion. Second, each transistor is modelled by an effective resistance whose value depends on the shape of the transistor,

the waveform driving its gate, and the load being driven by the transistor. For any given transistor, all of these factors are combined together into a single ratio that determines the effective resistance. The ratio approach means that small tables can be used to handle a large variety of actual situations.

Section 2 describes the stage decomposition and the advantages it provides over a logic-gate decomposition. Sections 3-5 present the three delay models. The lumped resistor-capacitor model of Section 3 is the simplest and fastest, but is only accurate to within about 25%. Section 4 describes the more complex slope model, which is usually accurate to within 10%. Section 5 applies the Penfield-Rubinstein models for distributed capacitance [7,9] to get still greater accuracy. Section 6 discusses the limitations of the models, and Section 7 compares this work to previous work in the area.

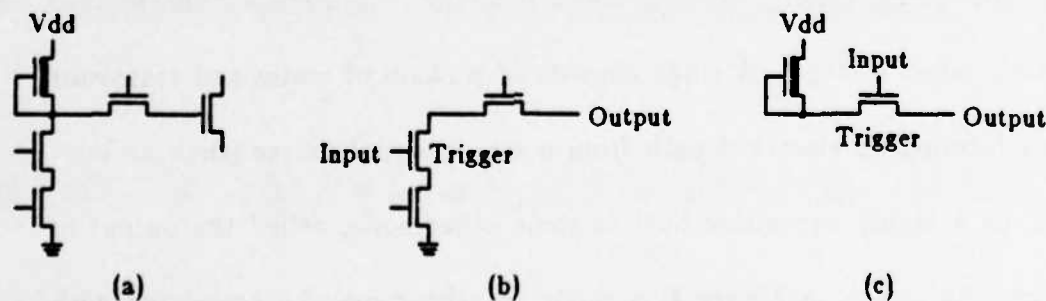
## 2. Stages

At any given time, Crystal's delay modeller considers a collection of transistors called a stage. A stage consists of a chain of nodes and transistor channels forming an electrical path from a strong signal source (such as  $V_{dd}$ , Ground, or a highly capacitive bus) to some other node, called the *output* of the stage. As shown in Figure 1, a single transistor may be associated with different stages during different phases of the analysis. Stages generally correspond to logic gates, except that pass transistors are lumped together

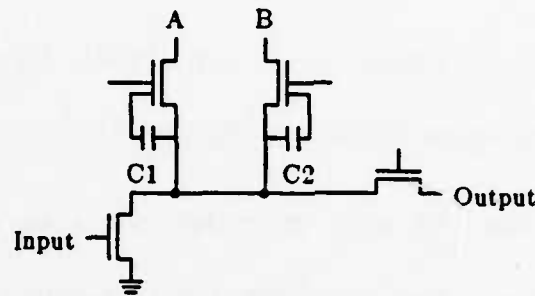
with the logic gates that drive them. See [5] for information on how stages are selected in Crystal.

The delay modeller is given the sizes and types of the transistors in the stage, and the parasitic resistances and capacitances of the nodes along the stage. One of the transistors is identified as the *trigger*: it is assumed to be the last transistor to turn on in the stage. Its gate is called the *input* of the stage, since it controls the activation of the stage. The modeller is also given information about the waveform at the input. Its job is to use this information to compute the waveform at the output of the stage, assuming that all the transistors in the stage except the trigger are turned on.

Waveforms are described at different levels of precision in different delay models. In the simplest delay model, each waveform is described by a single value: the time at which its voltage crosses the logic threshold (the input voltage for a standard inverter where the input and output voltages are equal in



**Figure 1.** A stage is a chain of transistors analyzed together for delay calculations. During different phases of analysis, the stages in (b) and (c) might be extracted from the circuit in (a) for delay calculation. The trigger transistor is the last one in the stage to turn on.



**Figure 2.** In computing delays, information from side paths is not included. For example, in this figure the capacitance at A and B is not included. However, the gate-source overlap capacitance from the side transistors (C1 and C2) is included in the parasitic capacitance of the stage.

the dc transfer curve). This is called the *inversion time* for the waveform. In the slope-based models each waveform is also characterized by its slope (in ns/volt) at the logic threshold voltage. This is called the *rise-time* of the waveform.

Crystal's delay modeller only considers information on the direct path between signal source and output. All side transistors connecting to the path are assumed to be turned off: their gate-source capacitance is included in the parasitic capacitance, but information on the far side of side transistors is ignored (see Figure 2). This approach is used in Crystal because the program does not have specific information about whether side transistors are turned on or off; if it automatically included all side capacitance, its delay estimates would be unrealistically high. Busses and pass transistor arrays account for most of the situations with many side paths, and in these cases only a single path is usually active through the structure at once. (If it becomes necessary

to include side paths, the results of Section 5 can be extended to handle them). Most stages in VLSI circuits are simple: in the critical paths found by Crystal, 80-90% of all stages contain only a single transistor.

Using stages as the basis for delay computation has worked out well, because both normal gates and pass transistors can be handled in a uniform fashion. Most previous timing analyzers have been organized around logic gates. They have generally had difficulty dealing with pass transistors because the non-linear pass transistor effects cannot be separated cleanly from the rest of the gate. The stage approach handles gates with or without pass transistor structures uniformly as chains of switches. It also accomodates complex gates such as AND-OR-INVERT.

### 3. The RC Model

The RC model is the simplest and least accurate of Crystal's models. It computes a resistance and capacitance value for each node and transistor along the stage. The resistances and capacitances are summed, and the product of these two lumped values is used as the delay time for the stage. No information about waveforms is used by the RC model: the delay time for the stage is added to the inversion time at the input to calculate the inversion time at the output.

Transistor Type	Ohms/square (transmitting 1)	Ohms/square (transmitting 0)
Enhancement	30000	15000
Enhancement driven by pass transistor	---	48000
Depletion Load	22000	---
Super-buffer (depletion, gate 1)	5500	5500
Depletion (gate 0 or unknown)	50000	7000

**Table I.** The effective resistances used by the RC model for nMOS circuits. The missing entries are for situations that do not ever occur (for example, depletion loads are used only to transmit 1's).

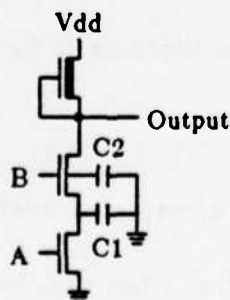
The effective resistance of each transistor in the stage is computed using a table based on the transistor's type (enhancement, depletion, etc.) and geometry. For each type of transistor there are two effective resistance values, each expressed in ohms per square. The first resistance value is used if the transistor is transmitting a logic one and the second is used if the transmitter is transmitting a logic zero. The values currently used in Crystal are given in Table I. The effective resistance of a transistor is computed by selecting the appropriate parameter value and multiplying it by the length/width ratio of the device.

The type of a transistor is not determined solely by its physical structure (enhancement, depletion, p-channel, etc.) but also by the way it is used in the circuit. For example, enhancement transistors driven through pass transistors have different characteristics than enhancement transistors driven directly by

depletion loads. When reading in circuits, Crystal distinguishes these two uses of enhancement transistors and use different types for each. Three different uses of depletion devices are also distinguished by Crystal, since they can result in different behavior. Users can add new types of their own and label transistors as being of the new types; this provides a crude facility for dealing with special circuit constructs such as bootstrap drivers.

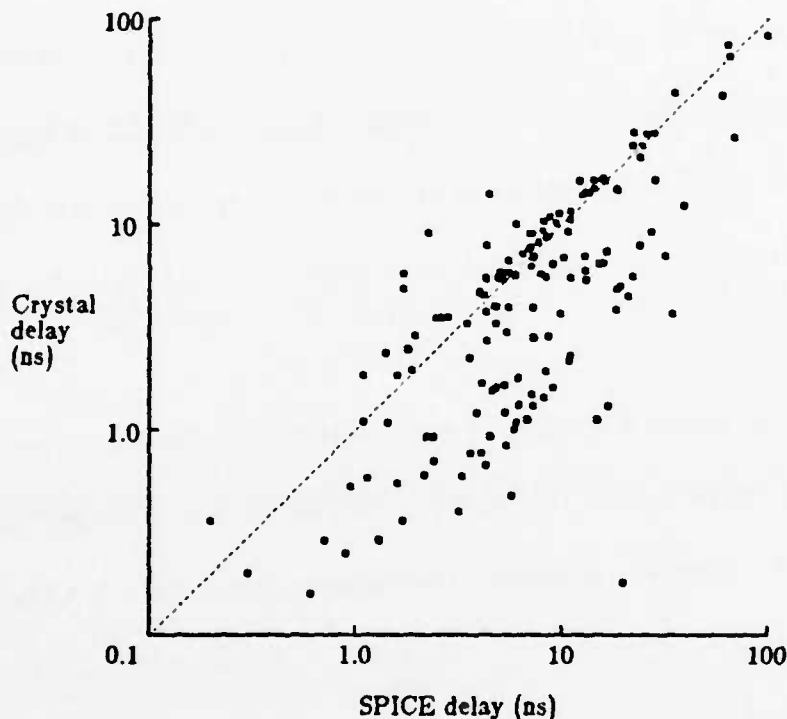
The table values are generated by running SPICE simulations of simple stages. In the SPICE simulations a step function is used to drive the input, and the effective resistance is computed by dividing the delay time by the load capacitance. It will be shown below that this results in an underestimation of effective resistance.

Capacitance includes parasitics from the nodes, gate-channel capacitance from transistors in the path, and gate-source capacitance from side transistors attached to the path. When summing the capacitances along the path, only



**Figure 3.** A switch-level approach to delay analysis automatically accounts for different delay characteristics at different inputs of a NAND gate. The capacitance at  $C1$  and  $C2$  will be included when calculating delays from A, but not from B.





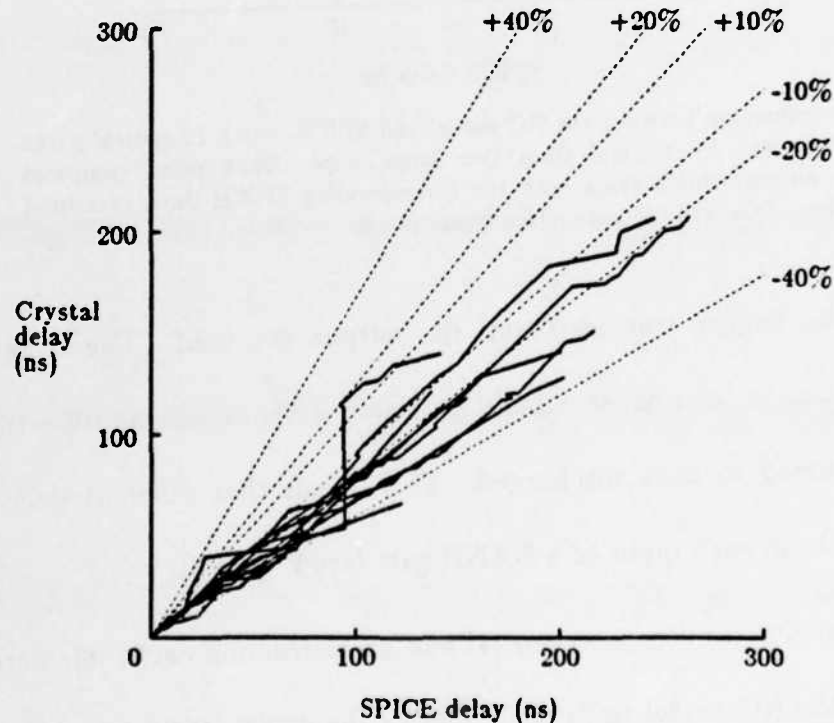
**Figure 4.** A comparison between the RC model and SPICE, using 12 critical paths (157 stages) extracted by Crystal from two large chips. Each point compares Crystal's delay estimate for a stage with the corresponding SPICE time, measured from a simulation of the critical path. Ideally, all points should fall along the diagonal.

those between the trigger transistor and the output are used. The trigger transistor is the last to turn on, so all the capacitance between it and the signal source is assumed to have discharged. This means that different delays will be computed from each input of a NAND gate (see Figure 3).

Two large circuits, a microprocessor [1] and an instruction cache [6], were used to compare the RC model to SPICE. Out of the 40000-50000 transistors in each chip, Crystal used the RC model to extract 12 critical paths containing a total of 157 stages. SPICE was used to simulate each critical path, and the

SPICE delays were compared with Crystal's estimates. Figure 4 makes a stage-by-stage comparison and Figure 5 compares total delays through the critical paths. Although the RC model often errs by a factor of 2 or more on individual stage calculations, the errors of successive stages tend to cancel. On average, it can usually estimate the total delay through a path to within 25% of SPICE.

Much of the RC model's error is due to consistent underestimation: the sum of all the RC delays is 20% less than the sum of all SPICE delays. This means that if all the effective resistances were simply scaled by 1.2, then the



**Figure 5.** A comparison between the RC model and SPICE, using total delays through critical paths. Each line corresponds to one critical path, and each point in the line corresponds to a stage. The point compares Crystal's and SPICE's estimates for the total delay in the critical path up through that stage.

RC delay estimates would generally fall within 10-15% of SPICE's estimates, at least for these test cases. Unfortunately, it isn't obvious whether or not such a scale factor depends on the circuit constructs or fabrication parameters, nor is it obvious how to characterize such a dependency if one exists. For these reasons, Crystal has not used the scale factor approach.

There are two sources of error in the RC model. One source of error is the lumping of resistances and capacitances. This tends to overestimate the delays since it assumes that all the capacitance must be discharged through all the resistance. Fortunately, most stages contain only a single transistor and small parasitic resistances, so lumping introduces only a small error and is not the major problem with the RC model. Section 5 will improve on the lumped model by applying the Penfield-Rubinstein models for distributed capacitance [7,9].

The second and most significant source of error in the RC model comes from its inability to deal with waveform shape. In practice, the effective resistance of a transistor depends on the waveform on its gate. If the trigger transistor turns on instantaneously, then its full driving power is used to drain the output capacitance and the transistor has a relatively low effective resistance. If the trigger turns on slowly, then it may do much or all of its work while only partially turned-on. In this case its effective resistance will be higher. In the extreme case of a very slow trigger, the output will settle as

quickly as the input changes, so the output voltage is determined by the input waveform and the dc transfer characteristics of the stage. The output will reach the logic threshold at exactly the same instant that the input reaches the logic threshold; since delays are defined at the logic threshold voltage, the stage will have zero delay and the transistor will have zero effective resistance.

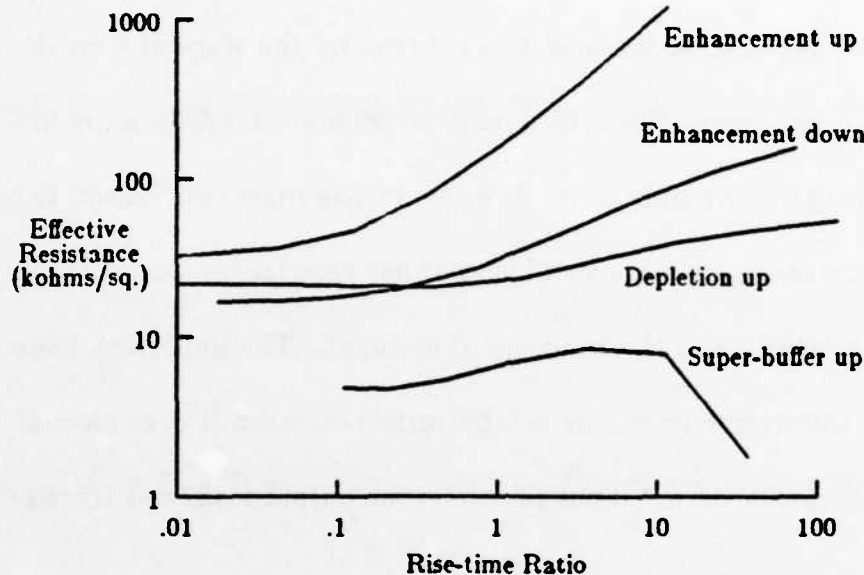
If all waveforms in a circuit have the same shape, then the effective resistances of transistors can be characterized using that waveform and the RC model will produce accurate results. Unfortunately, this is not the case in actual VLSI circuits. Although most of the waveforms have an exponential shape, they vary by more than three orders of magnitude in their slopes. As a result, the effective resistance of the transistors varies by more than a factor of ten and the RC model produces only a rough estimate for delays.

#### **4. The Slope Model**

The slope model incorporates information about waveform shape in order to make more accurate delay estimates. It assumes that all waveforms are exponential in overall shape but vary in their slopes. Each waveform is represented by its inversion time and its rise-time, in ns/volt at the logic threshold. A rise-time of zero corresponds to a step function, and a large rise-time corresponds to a slowly rising or falling signal.

Unfortunately, the effective resistance of a transistor depends not just on its gate's rise-time, but also on the load being driven by the stage and on the sizes of the transistors in the stage. If a stage is driving a large load, or has very small transistors, then only very slow input rise-times will affect the stage's delay. If a stage is driving a small load or has very large transistors, its delay will be more sensitive to the rise-time of its input. The important issue is whether or not the trigger transistor is fully turned-on when it does most of its work, and this depends on the input rise-time, the output load, and transistor size.

The key to implementing the slope model was the discovery that all of these factors can be combined into a single ratio, which alone determines the transistor's effective resistance. In the slope model, the output load and transistor size are characterized by the stage's *intrinsic rise-time*, which is the rise-time that would occur at the output if the input were driven by a step function. The input rise-time of a stage is then divided by the intrinsic rise-time of the stage's output to produce the *rise-time ratio* for the stage. An extensive series of SPICE simulations verified that, to a very close approximation, the effective resistance of a transistor is a function only of the *rise-time ratio*. This approximation holds across the entire range of structures that occur in our digital circuits. Pilling and Skalnik were apparently the first to suggest the ratio approach [8].



**Figure 6.** A plot of the tables that characterize the resistance of different types of transistors as a function of the rise-time ratio of the stage. A rise-time ratio of zero means the stage's input rises very quickly in comparison to the output. Different tables are used when transmitting zero (e.g. the "enhancement down" curve) and transmitting one (e.g. the "enhancement up" curve).

In the slope model, each transistor type is characterized by two resistance tables, one used when the transistor is transmitting a logic 0 and one used when it is transmitting a logic 1. Each table gives effective resistance values as a function of rise-time ratio. During timing analysis, the effective resistance of each trigger transistor is computed by interpolating in the appropriate table. All transistors except the trigger are assumed to be fully turned-on, so there is no need to interpolate for them (the effective resistance for the RC model, which is measured with a zero rise-time ratio, can be used).

The parameter tables for the slope model were generated in much the same way as for the RC model: SPICE simulations were run on simple stages

and parameters were extracted from the output. Each table contains six to ten values. Figure 6 plots the contents of a few of the tables for our nMOS process. As with the RC models, non-standard circuit structures can be handled by giving their transistors different types and generating special parameter tables for them.

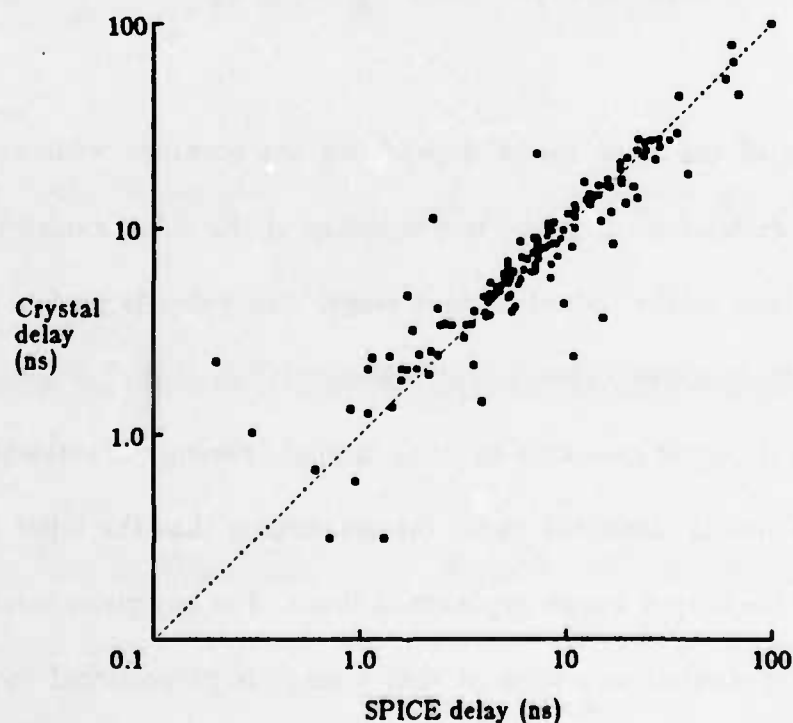
The ratio approach is important because it allows transistors to be parameterized with one-dimensional tables, instead of three-dimensional tables based on input rise-time, load, and transistor size. The three-dimensional approach would require large amounts of CPU time to generate the tables and would also make the delay analysis slower by requiring three-dimensional interpolation.

The accuracy of the slope model depends on the accuracy with which rise-times can be calculated. It is the responsibility of the delay modeller to estimate the rise-time at the output of each stage; this value is used as the input rise-time of later stages. The current implementation of the slope model approximates actual output rise-times by using intrinsic rise-times everywhere. The intrinsic rise-time is computed under the assumption that the input is a step function and the output has an exponential shape. For any given voltage, the slope of an exponential waveform at that voltage is proportional to the delay time to reach that voltage. This means that the intrinsic rise-time of a stage is proportional to its intrinsic delay, which is the delay computed by the

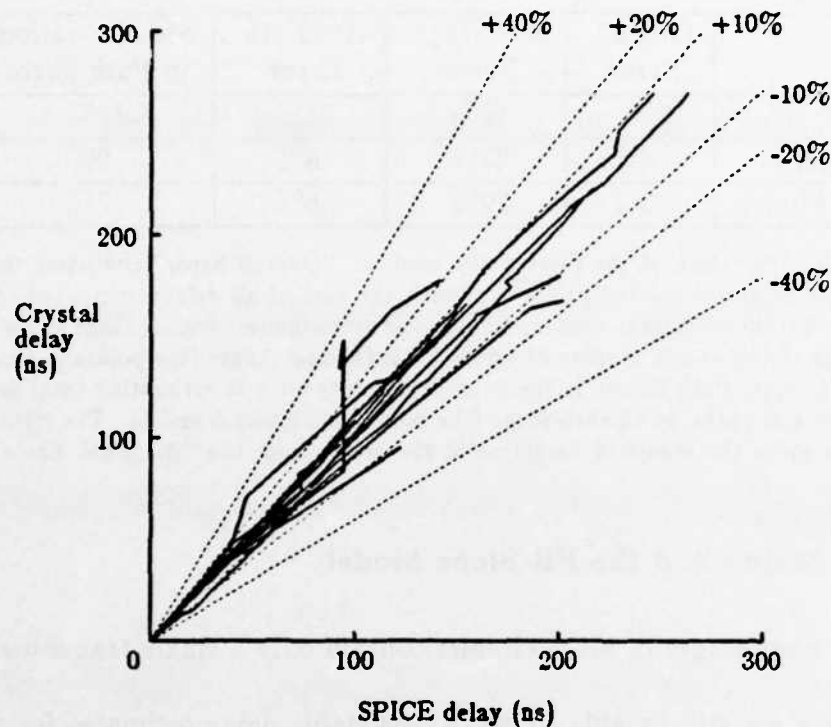


RC model.

The intrinsic rise-time is a lower bound on the actual rise-time of a stage's output: if the input has a large rise-time, then the actual rise-time at the output will be larger than the intrinsic rise-time. However, initial attempts at producing more accurate rise-time estimates did not produce noticeable improvements in the overall accuracy of the model. *(Note to referees: this rise-time work is still in progress, and I expect to evaluate it more thoroughly in time for the final conference version of the paper.)*



**Figure 7.** A stage-by-stage comparison between the slope model and SPICE, for the same critical paths as in Figure 4.



**Figure 8.** A total-delay comparison between the slope model and SPICE, using the same critical paths as in Figure 5.

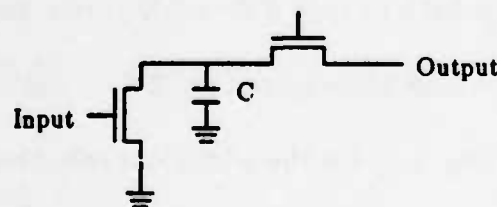
Figures 7 and 8 compare the slope model to SPICE with the same data as in Figures 4 and 5. These figures show that the slope model is substantially more accurate than the RC model. The average error for individual stages was reduced from 45% to 23%, and the average overall error over several consecutive stages dropped from 24% to only 8%. Only rarely does the estimate for a critical path differ from SPICE by more than 20%. Table II summarizes this data and the corresponding data for the additional enhancement described in Section 5.

Model	Overall Error	Av. Stage Error	Av. Path Error	Std. Deviation in Path Error
RC Model	-22%	45%	24%	10%
Slope Model	4%	23%	8%	9%
PR-Slope Model	2%	20%	6%	7%

**Table II.** A comparison of the three delay models. "Overall Error" compares the sum of all the delays computed by Crystal with the sum of all delays computed by SPICE, to point out consistent underestimates or overestimates. "Av. Stage Error" is the average of the absolute value of errors for individual stages (the points in Figures 4 and 7). "Av. Path Error" is the average absolute error in estimating total delays in the critical paths up to each stage (the points in Figures 5 and 8). The right-most column gives the standard deviation in the errors from the "Av. Path Error" column.

## 5. Complex Stages and the PR-Slope Model

Although most stages in MOS circuits contain only a single transistor, the delay modeller must still be able to make reasonable delay estimates for more complex stages. In the RC and slope models, complex stages are handled by lumping resistances and capacitances. The lumped approach is pessimistic for complex paths with distributed capacitance, since it assumes that all the capacitance must be discharged through all the resistance (see Figure 9 for an example).



**Figure 9.** The basic slope model will compute delays under the assumption that the capacitance at C must discharge through both transistors, whereas in fact it only discharges through one.

In order to handle such situations more accurately, a new model called *PR-slope* was implemented. It is similar to the slope model except that it uses the results of Penfield and Rubinstein [7,9] to avoid lumping the capacitance. [9] provides upper and lower bounds for the delay; Crystal uses the average of the two. Since each stage is a linear chain of transistors with no side trees, the average of the upper and lower bounds is

$$delay = \sum_i R_i C_i$$

$R_i$  is the resistance from the signal source to point  $i$ , and  $C_i$  is the capacitance at point  $i$ . This means that instead of weighting all the capacitance by all the resistance, each separate capacitor is weighted only by the resistance between it and the signal source.

The addition of the Penfield-Rubinstein models made an additional improvement in accuracy, which is summarized in Table I. The average error for a single-stage estimate dropped from 23% to 20%, and the average error over paths containing several stages dropped from 8% to 6%. The small improvement supports the conclusion that complex stages are rare in large circuits and suggests that the main remaining cause of error is inaccuracy in estimating rise-times. However, the PR-slope model is almost as efficient as the basic slope model, and can avoid gross over-estimates that will occasionally happen in the basic slope model.

## 6. Limits of the Models

The models presented here are nearly as accurate as SPICE over a wide range of digital circuit constructs and loading characteristics. There are three limitations to the models, however. The first, inaccuracy in estimating rise-times, was discussed in Section 4. The other two limitations are discussed in the paragraphs below. They have to do with complex stages and the number of transistor types.

Although the PR-slope model makes a first-order attempt to deal with complex stages, there are still several situations that can cause it to produce inaccurate results. One source of error in complex stages is the assumption that only a single transistor is turning on or off at once. If two transistors turn on simultaneously in a NOR gate, the gate's delay will be less than predicted; if two transistors turn on simultaneously (and slowly) in a NAND gate, its delay will be greater than predicted. Another source of error in complex stages is the additive treatment of different transistors and resistors: total resistances and rise-times for stages are computed by summing the contributions of each device along the stage. Although this is an accurate approximation for resistors, it is less accurate for non-linear devices such as pass transistors.

The third potential limitation of the models has to do with the number of transistor types. When transistors are used in different ways, such as

bootstrap drivers, super buffers, or even gates with different pullup/pulldown ratios, they must be modelled with separate transistor types. In general, each construct with a separate dc transfer characteristic must be modelled separately. SPICE simulations must be run to extract the parameters for each type. In principle this allows an unlimited number of different circuit constructs: every transistor in the circuit could ostensibly be given a different type. However, the number of different transistor types is limited in practice by the human and computer time that must be expended to extract all their parameters. If a large number of transistor types is required to model a circuit accurately, too much parameter extraction time will be required for the approach to be reasonable. Fortunately, our current design style at U.C. Berkeley is uniform enough that a half dozen different transistor types is enough.

## 7. Related Work

The importance of using waveform information in computing delays has been recognized for some time [4,8,10,11]. In [11], analytic models were developed to model the effects of waveforms and those models were validated against circuit simulation, but only over a relatively small range of rise-time ratios. As a consequence, Tokuda et al. concluded that load transistors were insensitive to waveform. Tamura et al. have also developed an analytical

model for waveform effects [10], but their model appears to apply only to simple gates without pass transistors. The ratio approach to modelling waveform effects was suggested in 1972 by Pilling and Skalnik [8], although they also did not deal with pass transistors, and the idea doesn't appear to have been used since then.

## 8. Conclusions

This paper shows that simple models can estimate delays in MOS digital circuits to within 10% of SPICE. Since other factors, such as variations in processing, are likely to cause errors at least as great as this, the accuracy of the simple models appears to be acceptable for a wide variety of applications. The models are fast: delays can be estimated for typical stages in less than one millisecond using the PR-slope model.

Two factors contribute to the success of Crystal's models. The first factor is the switch-level approach based on stages: it enables the system to handle a variety of different circuit constructs in a uniform fashion. The second factor is the use of rise-time ratios, which results in small parameter tables and fast interpolation.



## 9. Acknowledgements

Gordon Hamachi, Bob Mayo, Walter Scott, Carlo Séquin, and George Taylor all provided helpful comments on early drafts of this paper. The work was supported in part by the Defense Advanced Research Projects Agency under contract N00034-K-0251.

## 10. References

- [1] Katevenis, M., Sherburne, R. Patterson, D., and Séquin, C.S. "The RISC II Micro-Architecture." *Proc. IFIP TC10/WG10.5, International Conference on VLSI*, North Holland, 1983, pp. 349-359.
- [2] Lelarasmee, E. and Sangiovanni-Vincentelli, A. "RELAX: A New Circuit Simulator for Large Scale MOS Integrated Circuits." *Proc. 19th Design Automation Conference*, 1982, pp. 682-691.
- [3] Nagel, L.S. "SPICE2: A Computer Program to Simulate Semiconductor Circuits." ERL Memo ERL-M520, University of California, Berkeley, May 1975.
- [4] Okazaki, K., Moriya, T. and Yahara, T. "A Multiple Media Delay Simulator for MOS LSI Circuits." *Proc. 20th Design Automation Conference*, 1984. pp. 279-285.
- [5] Ousterhout, J.K. "Crystal: A Timing Analyzer for nMOS VLSI Circuits." *Proc. Third Caltech Conference on VLSI*, R. Bryant, ed., Computer

Science Press, 1983, pp. 57-70.

- [6] Patterson, D.A., et al. "Architecture of a VLSI Instruction Cache for a RISC." *Proc. 10th International Symposium on Computer Architecture*, 1983 (*SIGARCH Newsletter*, Vol. 11, No. 3), pp. 108-116.
- [7] Penfield, P. Jr. and Rubinstein, J. "Signal Delay in RC Tree Networks." *Proc. 18th Design Automation Conference*, 1981, pp. 613-617.
- [8] Pilling, D.J. and Skalnik, J.G. "A Circuit Model for Predicting Transient Delays in LSI Logic Systems." *Proc. 6th Asilomar Conference on Circuits and Systems*, 1972, pp. 424-428.
- [9] Rubinstein, J., Penfield, P. Jr., and Horowitz, M.A. "Signal Delay in RC Tree Networks." *IEEE Transactions on CAD/ICAS*, Vol. CAD-2, No. 3, July 1983, pp. 202-211.
- [10] Tamura, E., Ogawa, K. and Nakano, T. "Path Delay Analysis for Hierarchical Building Block Layout System." *Proc. 20th Design Automation Conference*, 1984, pp. 411-418.
- [11] Tokuda, T., et al. "Delay-Time Modeling for ED MOS Logic LSI." *IEEE Transactions on CAD/ICAS*, Vol. CAD-2, No. 3, July 1983, pp. 129-134.

# Magic: A VLSI Layout System

John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo,  
Walter S. Scott, and George S. Taylor

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California  
Berkeley, CA 94720

## Abstract

Magic is a "smart" layout system for integrated circuits. It incorporates expertise about design rules and connectivity directly into the layout system in order to implement powerful new operations, including: a continuous design-rule checker that operates in background to maintain an up-to-date picture of violations; an operation called *plowing* that permits interactive stretching and compaction; and routing tools that can work under and around existing connections in the channels. Magic uses a new data structure called *corner stitching* to achieve an efficient implementation of these operations.

**Keywords and Phrases:** interactive layout editor, corner stitching, design-rule checking, routing, stretching, compaction.

## 1. Introduction

Magic is a new interactive layout editing system for large-scale MOS custom integrated circuits. The system contains knowledge about geometrical design rules, transistors, connectivity, and routing. Magic uses its knowledge to provide powerful interactive operations that simplify the task of creating layouts. Moreover, once a layout has been entered, Magic makes it easy to modify it; this permits designers to fix bugs easily, experiment with alternative designs, and make performance enhancements.

Magic provides several new operations for its users. Design rules are checked continuously and incrementally during editing sessions to keep up-to-date information about violations. When the layout is finished, then so is the design-rule check. A new operation called *plowing* allows layouts to be compacted and stretched while observing all the design rules and maintaining circuit structure. Routing tools are provided that can work under and around existing wires in the channels (such as power and ground routing) while still providing the traditional efficiency of a channel router.

Two aspects of Magic's implementation make the new operations possible. First, the system is based on a data structure called *corner stitching* which is both simple and efficient for a variety of geometrical operations [6]. Without corner stitching, most of Magic's new operations would be too slow for interactive use. Second, designs in Magic are specified using *abstract layers*, rather

than actual mask layers. The abstract layers represent circuit structures such as contacts and transistors in a form that appears somewhat like sticks [14] except that objects are seen in their actual sizes and positions. The abstract layers incur no density penalty, but they simplify the designer's view of the system and provide more explicit information about the circuit structure.

This paper gives an overview of the Magic system. Section 2 describes the specific problems Magic attempts to solve, and the overall approach of the system. Sections 3 and 4 describe the data structure and abstract layers used in the Magic implementation. Sections 5-11 discuss Magic's new operations, and Section 12 presents the implementation status of the system. Three additional papers in this technical report discuss design-rule checking, plowing, and routing in detail [2,11,12].

## **2. Background and Goals**

Our previous layout editing systems, Caesar [5,7] and KIC2 [3], have been used since 1980 for a variety of large and small designs in several MOS technologies. They are similar to systems currently in use in industry. Although our systems have proven quite useful, we uncovered a few areas where they (and most other existing layout systems) are inadequate. The most severe inadequacy is in the area of routing, where most systems provide little support. We estimate that between 25% and 50% of all layout time for our circuits is used

for hand-routing the global interconnections, even though the circuits are highly regular to begin with. The task of routing is tedious and error-prone.

A more general problem is one of flexibility. Once a design has been entered into the layout system, it is hard to change. This makes it difficult to fix bugs found late in the layout process, and almost impossible to experiment with alternative designs. If designers cannot experiment with and evaluate alternatives, it is hard for them to develop intuition about what is good and bad. Routing is the most extreme example of the flexibility problem. It takes so long to route a circuit that it is out of the question to re-route a chip to try a new floor-plan. Even small cells are difficult to change: modest changes to the topology of a cell often require the entire cell to be re-entered. In many industrial settings, layouts are so difficult to enter and modify that designs are completely frozen before layout begins.

Our overall goal for Magic is to increase the power and flexibility of the layout editor so that designs can be entered quickly and modified easily. When the system is complete, we hope it will provide order-of-magnitude speedups for three different parts of the design process:

- 1) Once a large circuit has been routed, it should be possible to remove the routing and re-route in a few hours. Even the initial routing should not require more than a few days for a large custom circuit. With our current systems, routing requires a few weeks to a few months.

- 2) The turnaround time for small bug fixes should be less than 15 minutes. For example, if a bug is found while simulating the circuit extracted from a layout, it should be possible to fix the layout, verify that the new layout meets the design rules, and re-extract the circuit, all in 15 minutes. This process currently requires several hours of CPU time and at least a half-day of elapsed time.
- 3) It should not take more than 30 seconds to 1 minute to re-arrange a cell to try out a different topology. With our current systems this requires anywhere from tens of minutes to several hours.

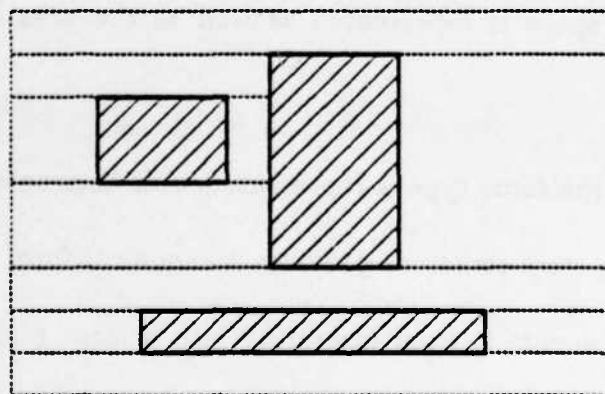
Magic meets these goals by combining circuit expertise with an interactive editor. It understands layout rules; it knows what transistors and contacts are (and that they must be treated differently than wires); and it knows how to route wires efficiently. Magic uses the circuit knowledge to provide interactive operations that re-arrange a circuit *as a circuit*, rather than as a collection of geometrical objects. It also performs analysis operations, like design-rule checking, *incrementally*, as the circuit is created and modified. This means that only a small amount of work must be done each time the circuit is modified.



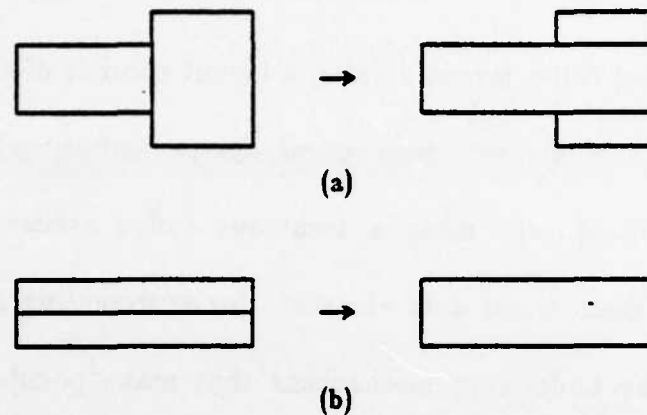
### 3. Corner Stitching

In Magic, as in most other layout editors, a layout consists of cells. Each cell contains two sorts of things: geometrical shapes and subcells. Magic represents the contents of cells using a technique called *corner stitching*. Corner stitching is a geometrical data structure for representing Manhattan shapes. It provides the underlying mechanisms that make possible most of Magic's advanced features. Corner stitching is simple, provides a variety of efficient searching operations, and allows the database to be modified quickly. What follows is a brief introduction to corner stitching. See [6] for a more complete description.

The basic elements in corner stitching are *planes* and *tiles*. Each cell contains a number of corner-stitched planes to represent the cell's geometries



**Figure 1.** Every point in a corner-stitched plane is contained in exactly one tile. In this case there are three solid tiles, and the rest of the plane is covered by space tiles (dotted lines). The space tiles on the sides extend to infinity. In general, a plane may contain many different types of tiles.



**Figure 2.** Areas of the same type of material are represented with horizontal strips that are as wide as possible, then as tall as possible. In each of the figures the tile structure on the left is illegal and is converted into the tile structure on the right. In (a) it is illegal for two tiles of the same type to share a vertical edge. In (b) the two tiles must be merged together since they have exactly the same horizontal span.

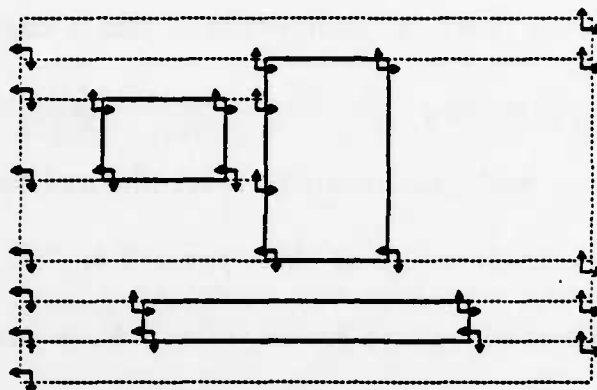
and subcells; each plane consists of a number of rectangular tiles of different types. There are three important properties of a corner-stitched plane, illustrated in Figures 1, 2, and 3:

**Coverage:** Each point in the x-y plane is contained in exactly one tile (Figure

1). Empty space is represented as well as the area covered with material.

**Strips:** Material of the same type is represented with horizontal strips (Figure 2). The strip structure provides a canonical form for the database and prevents it from fracturing into a large number of small tiles.

**Stitches:** Tiles are linked together at their corners. Each tile contains four of these links, called *stitches* (Figure 3).



**Figure 3.** Each tile is linked to its neighbors with four pointers, called *corner stitches*. The corner stitches provide a form of two-dimensional sorting. They permit a variety of geometrical operations to be performed efficiently, such as searching an area or finding all the neighboring tiles on one side of a given tile.

The stitches permit a variety of search operations to be performed efficiently, including: finding the tile containing a given point; finding all the tiles in an area; finding all the tiles that are neighbors of a given tile; and traversing a connected region of tiles. The coverage property makes it easy to update the database in response to edits, and the strip property keeps the database representation small. To the best of our knowledge, corner stitching is unique in its ability to provide these efficient two-dimensional searches and yet permit fast updates of the kind needed in an interactive tool. The only disadvantage of corner stitching in comparison to less powerful data structures is that it requires more storage space (about three times as much space as structures based on linked lists of rectangles). Even so, the storage requirements do not appear to be a problem for chips likely to be designed in the next several years.

#### 4. Abstract Layers

There are several ways in which corner-stitched planes might be used to represent the mask geometries in a cell. One alternative is to use a separate plane for each mask layer; each plane contains space tiles and tiles of one particular mask type. The disadvantage of this approach is that many operations, such as design-rule checking and circuit extraction, require information about layer interactions (such as polysilicon crossing diffusion to form a transistor, or implants changing the type of a transistor). With a separate plane per mask layer, these operations will spend a substantial amount of time cross-registering the information on different planes.

Another alternative is to place all the mask layers into a single corner-stitched plane. Since there can be only one tile at a given point in a given plane, different tile types must be used for each possible overlap of mask layers. This eliminates the registration problem, but results in a large number of small tiles where several mask layers overlap. Even though many of the layer overlaps are not significant (such as metal and implant), separate tile types have to be used to represent them. As a result, the database fragments into a large number of tiles, and the overheads for all operations increase.

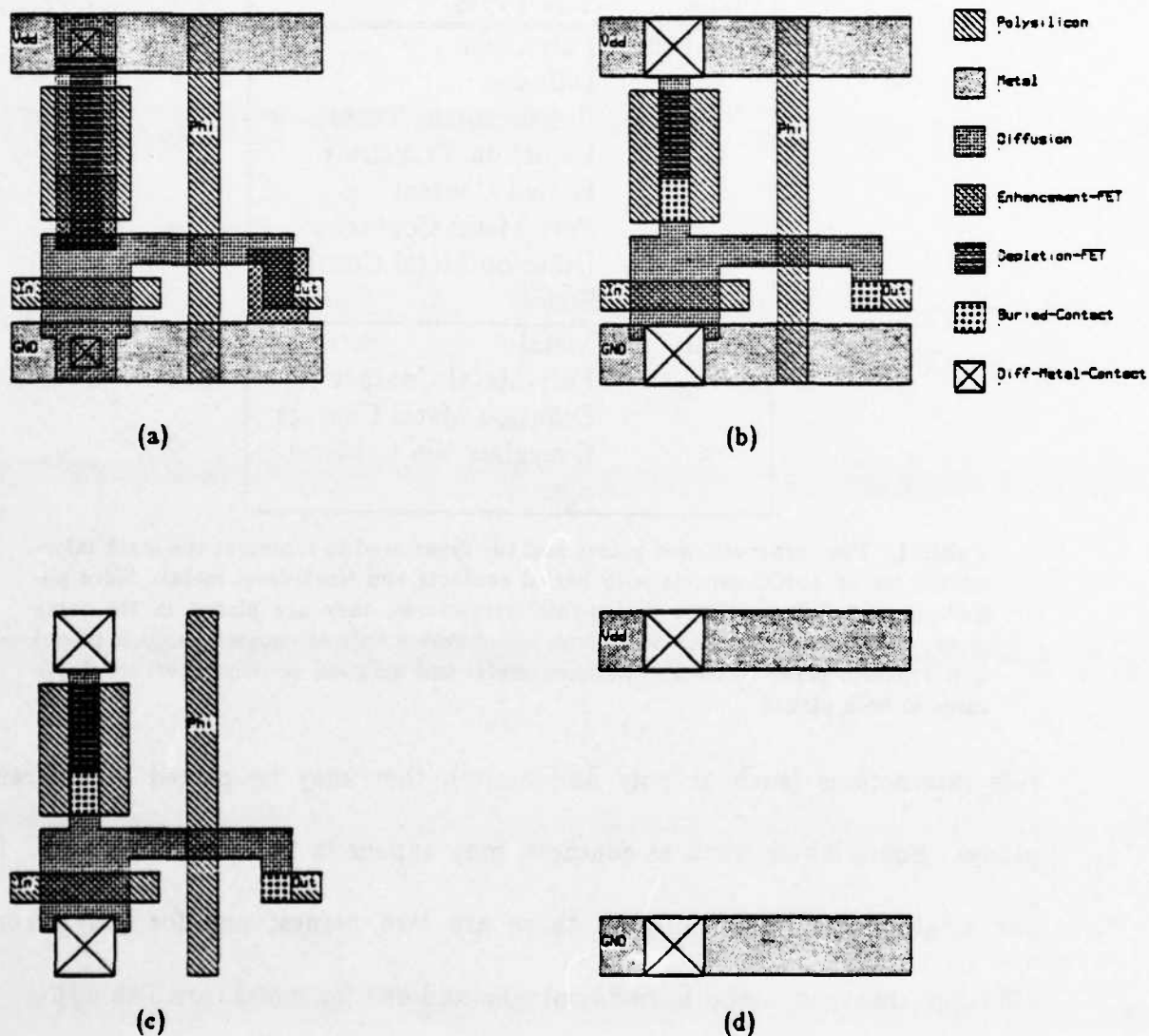
The solution we chose for Magic lies between these two extremes. We decided to use a small number of planes, where each plane contains a set of layers that have design-rule interactions. If layers do not have direct design-

Plane	Tile Types
Poly-Diff	Polysilicon
	Diffusion
	Enhancement Transistor
	Depletion Transistor
	Buried Contact
	Poly-Metal Contact
	Diffusion-Metal Contact
	Space
Metal	Metal
	Poly-Metal Contact
	Diffusion-Metal Contact
	Overglass Via to Metal
	Space

**Table I.** The corner-stitched planes and tile types used to represent the mask information for an nMOS process with buried contacts and single-level metal. Since polysilicon and diffusion have design-rule interactions, they are placed in the same plane. Metal interacts with polysilicon and diffusion only at contacts, so it is placed in a separate plane. Contacts between metal and diffusion or polysilicon are duplicated in both planes.

rule interactions (such as poly and metal), they may be placed in different planes. Some layers, such as contacts, may appear in two or more planes. In our single-metal nMOS process there are two planes: one for polysilicon, diffusion, transistors, and buried contacts; and one for metal (see Table I).

We also decided not to represent every mask layer explicitly. Instead of dealing with actual mask layers, Magic is based around *abstract layers*. The abstract layers do not include implants, wells, buried contact windows, or contact vias. Instead, the abstract layers include separate tile types for each possible kind of transistor and contact. Magic generates the missing mask layers when it creates CIF files for fabrication. Table I gives the planes and abstract



**Figure 4.** In Magic, transistors and contacts are drawn in an abstract form: (a) a three-transistor shift-register cell, showing actual mask layers; (b) the same cell as it is seen in Magic; (c) the information in Magic's poly-diff plane; (d) the information in Magic's metal plane. Contacts are duplicated in each plane.

layers used in Magic, and Figure 4 illustrates how the abstract layers are used in a sample cell. Abstract layers change the way a circuit looks on the screen, but they do not incur any density penalty.

The Magic design style is similar to sticks and symbolic systems such as Mulga [13] and VTVID [10], except that the geometries are fully fleshed. Designers draw the primary interconnection layers and simplified forms of contacts and transistors. Magic fills in the structural details. As in sticks, there are simple operations for stretching and compacting cells. The advantage of Magic's abstract-layer approach is that designers can see the exact size and shape of a cell while it is being edited, and they only work with a single representation of the circuit. When using sticks, designers go back and forth between the sticks and mask representation; the final size of the cell is hard to determine until it has been compacted and fleshed out. The following sections will show how the abstract layers simplify design-rule checking, plowing, and circuit extraction.

In addition to the planes used to hold mask geometry, each cell contains another plane to hold information about its subcells. Subcells are allowed to overlap in Magic; each distinct subcell area or overlap between subcells is represented with a different tile in the subcell plane. Each tile contains pointers to all of the subcells that cover the tile's area. By using corner-stitching in this way, it is easy to find subcell interactions and to determine which (if any) subcells cover a particular area.



## 5. Basic Commands

The basic set of commands in Magic is similar to the commands in Caesar [5,7]. Mask geometry is edited in a style like painting: a rectangle is placed over an area of the layout, and mask layers may be painted or erased over the area of the rectangle. Additional operations are provided to make a copy of all the "paint" in a rectangular area and copy it back at a different place in the layout. The corner-stitched representation is invisible to users.

Magic also provides commands for manipulating subcells. Subcells may be placed in a parent, moved, mirrored in x or y, rotated (by multiples of 90 degrees only), arrayed, and deleted. Subcells are handled by reference, not by copying: if a subcell is modified, the modifications will be reflected everywhere that the subcell is used.

## 6. Incremental Design-Rule Checking

Design-rule checking is an integral part of the Magic system. Our main goal was to make the checker very fast, particularly for small changes: the cost of reverifying a layout should be proportional to the amount of the layout that has been changed, not to the total size of the layout. To achieve this, Magic's design-rule checker runs continuously in the background during editing sessions. When the layout is changed, Magic records the areas that must be reverified. The design-rule checker then rechecks these areas during the

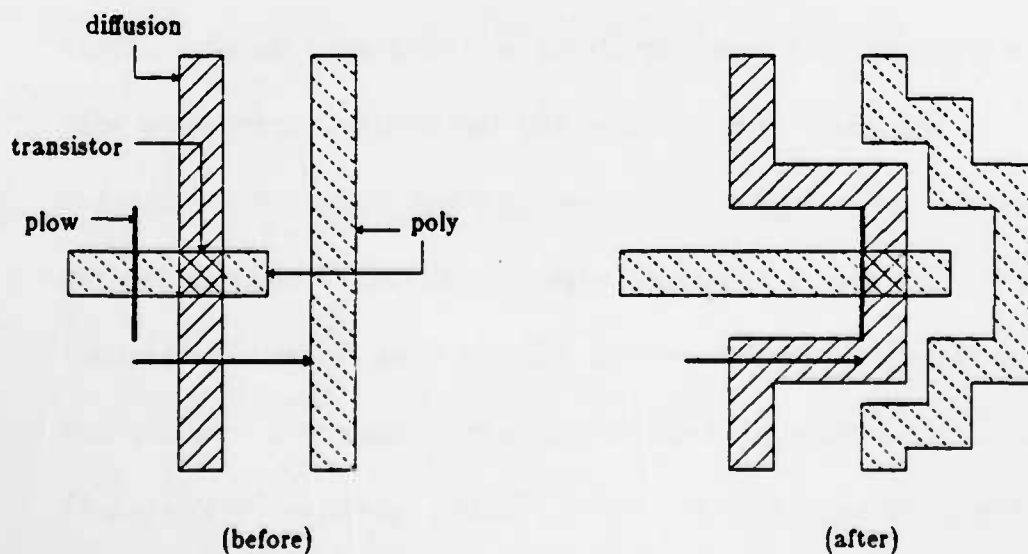
time when the user is thinking. For small changes, error information appears on the screen instantly (and also disappears instantly when the problem has been fixed). For large changes (such as moving one large subcell on top of another), it may take seconds or minutes for the design-rule checker to complete its job. In the meantime, the designer can continue editing. If reverification hasn't been completed when an editing session ends, the areas still to be reverified are stored with the cell so that reverification can be completed the next time the cell is edited. Error information is also stored with cells until the errors are fixed. With this mechanism, there is never a need to check a layout "from scratch."

Magic's basic rule-checker works from the edges in a design. Based on the type of material on either side of an edge, it verifies constraints that require certain layers to be present or absent in areas around the edge. There are several reasons why corner stitching and the abstract layers allow edge rules to be checked quickly. Each corner-stitched plane can be checked independently. All the "interesting" edges are already present in the tile structure, so there is no need to register different mask layers. The abstract layers make it unnecessary to check formation rules associated with implants and vias. Lastly, corner stitching provides efficient algorithms for locating all the edges in an area and for searching the constraint areas.

In addition to a fast basic checker, the incremental rule checker contains algorithms for handling hierarchy. When a cell in the middle of a hierarchical layout is changed, Magic checks interactions between this cell and its subcells, and also interactions between this cell and other cells in its parents and grandparents. More details on the basic DRC mechanism and on Magic's hierarchical approach can be found in [12].

## 7. Plowing

Plowing is a simple operation that can be used to rearrange a layout without changing the electrical circuit that it represents. To invoke the plow operation, the user specifies a vertical or horizontal line segment (the *plow*) and a distance perpendicular to it (the *plow distance*). See Figure 5. Magic



**Figure 5.** In plowing, a horizontal or vertical line is moved across the circuit, pushing material out of its way. Design rules and connectivity are maintained.

sweeps the plow for the specified distance, and moves and moves all material out of the area swept out by the plow. The edges of this material are likewise treated as plows, pushing other material in front of them. Mask geometry in front of the plow is compacted as it is moved, and mask geometry that crosses the initial position of the plow is stretched behind the plow. Jogs are inserted at the ends of the plow. The plow operation maintains design rules and connectivity so that it doesn't change the electrical structure of the circuit. Most material, such as polysilicon, diffusion, and metal, may be stretched or compacted by plowing; transistors and contacts may be moved, but their shape will not change.

Plowing provides all the operations of a sticks-based system, while still working with fully-fleshed geometry. If a large plow is placed to one side of a cell and then moved across the cell, the cell will be compacted. If a large plow is placed across the middle of the cell and moved, the cell will be stretched at that point. A small plow placed in the middle of a cell can be used to open up empty space for new transistors or wiring. Plowing may be used both on low-level cells containing only geometry, and on high-level cells containing subcells and routing. Plowing moves each subcell as a unit, without affecting the contents of the subcell.

The implementation of plowing is dependent on corner stitching, abstract layers, and the edge-based design rules. Corner stitching provides the fast

geometric operations used to search out plow areas. The abstract layers tell Magic about materials that cannot be stretched or compacted (such as transistors). The edge-based design rules indicate what must be moved out of the way when a particular edge of material is moved. By working from the same data structure used for editing and design-rule checking, the plowing operation avoids the overhead of converting between representations. See [11] for a detailed presentation of the plowing operation and its implementation.

## **8. Circuit Extraction and Cell Overlaps**

The Magic database makes circuit extraction almost trivial for individual cells. Because of the abstract layers and corner stitching, the circuit is almost completely extracted to begin with. All that is needed is to traverse the tile structure and record information about what connects to what. There is no need to register layers or infer the structure and type of transistors and contacts: all this information is represented explicitly.

For hierarchical designs, the situation is complicated when cells overlap. Each cell uses a separate set of corner-stitched planes, so information from the separate planes must be combined in order to find out what connects to what. If arbitrary overlaps are allowed, then transistors may be split between cells, or may be formed or broken by cell overlaps. In this case, circuits cannot be extracted hierarchically, since the structure of a cell may be changed by the

way it is used in its parents.

One approach to the overlap problem is to prohibit cell overlaps. This has two drawbacks, however. First, it makes for clumsy designs, since overlap areas must be placed in separate cells. This makes it harder to understand designs and harder to re-use cells. Second, it doesn't eliminate the problems in circuit extraction, since information will still have to be registered along the boundaries of abutting cells. For example, a cell abutment can cause two separate transistors to join together.

Instead of prohibiting overlaps, we decided to restrict them. In Magic, cells may abut or overlap as long as this only connects portions of the cells without changing their transistor structure. Overlaps and abutments may not change the type or number of transistors from what it would be without the overlap (e.g. polysilicon from one cell may not overlap diffusion from another cell, since this would create a new transistor). These restrictions can be verified by using a special set of design rules in the part of the design-rule checker that deals with cell overlaps.

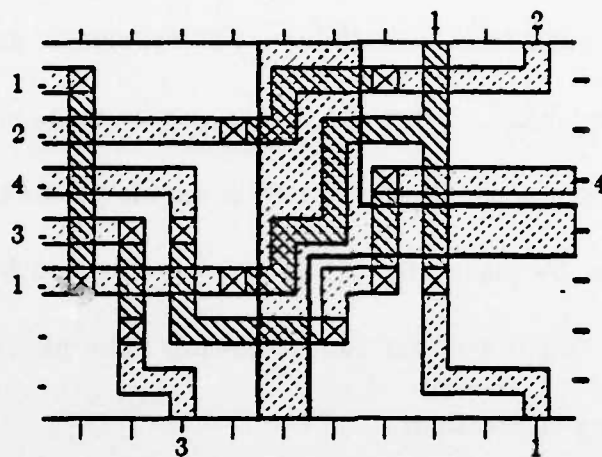
Our solution still requires information to be registered between subcells, but it allows the extracted circuit to be represented (and extracted) hierarchically. The extracted circuit for any cell consists of the circuits of its subcells, plus the circuit of the cell itself, plus a few connections between the subcells.

## 9. Routing

Routing is the single most important area where we hope Magic will speed up the design process. Most of the Magic routing effort has been spent in two areas: a) creating a channel router that can work around obstacles in the channel (such as previously-placed interconnections and power and ground routing); and b) developing an interface between grid-based routers and non-gridded custom designs.

Magic uses a standard three-phase approach to routing. In the first phase, called *channel decomposition*, the empty space of the layout is divided up into rectangular channels. In the second phase, called *global routing*, nets are processed sequentially to decide which channels will be crossed by each. In the third phase, called *channel routing*, each channel is considered separately and wires are placed to achieve the necessary connections within the channel. Magic's channel decomposer (which is not yet implemented) will be based on the bottleneck approach of the BBL system [1]. Global routing (also not yet implemented) will use a standard wavefront approach [4]. Both of these will use corner-stitching to keep track of the channel space. The channel router has been implemented, and is an extended version of Rivest's greedy router [9]. Magic does not provide placement tools: in our design style, placement is an important architectural decision and must be handled by designers.



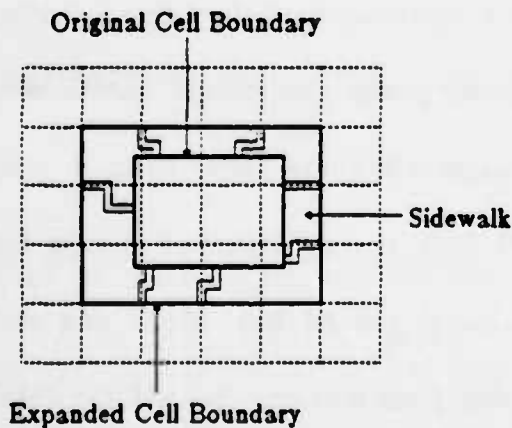


**Figure 6.** An example of routing with a single-layer obstacle in the channel. The router tries to avoid the thickest part of the obstacle if possible.

In order to make the routing tools useable in a custom design environment, we have developed a channel router that can work around obstacles in the channels. It is important for designers to be able to wire critical nets by hand, and to have the automatic routing tools route the less critical nets without affecting the hand-routed ones. It is also convenient to run power and ground routing tools as a separate step before signal routing, and have the signal router work around the power and ground wires. Where there are obstacles in the channels, Magic will route under them if possible, and will route around those that block both routing layers. For very large obstacles in one layer, such as a wide metal ground bus, Magic can make interconnections under the obstacles using river-routing. See [2] for details on how Rivest's greedy router has been extended to handle obstacles. Figure 6 shows an example of results produced by the Magic router.

The evasive router, combined with plowing and the other editing features, provides designers with considerable flexibility. Critical signals and power and ground can be routed by hand. Then the router can be invoked to complete the rest of the interconnections. If the router is unable to make all connections, the final ones can be placed by hand. Or, plowing can be used to rearrange the placement and the router can be re-run. The plowing operation will maintain the existing connections.

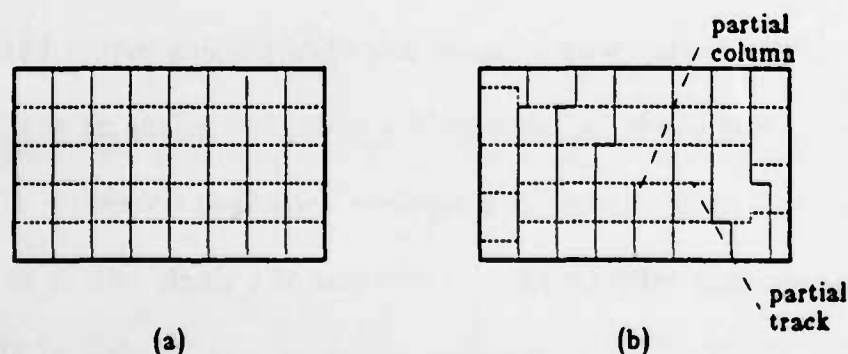
We have also extended the standard routing approach to handle designs that are not based on a uniform routing grid. Most channel routers assume a uniform grid based on the minimum wire spacing: channel dimensions must be an integral number of grid units, and all wires must enter and leave channels on grid points. Unfortunately, custom cells are not usually designed with the router's grid in mind, so the cell boundaries and terminals do not line up on a



**Figure 7.** In the sidewalk approach, each cell is enlarged so that its boundary is grid-aligned. Then connections on the edge of the original cell are routed to grid points on the outside of the sidewalk.

master grid. We are experimenting with two approaches to this problem, called *sidewalks* and *flexible grid*.

The sidewalk approach is illustrated in Figure 7, and involves a pre-routing step where all cells are expanded so that their dimensions are integral grid units. This additional cell area is called its *sidewalk*. In addition, wires are added to connect the terminals of the cell to grid points on the outer edge of the sidewalk. After the sidewalk generation stage, everything is grid aligned so standard routing tools can be used. Magic currently implements the sidewalk approach. Sidewalks are inefficient because the sidewalk areas cannot be used for channel routing, even though they usually contain little material. Sidewalks typically cause the channels to be reduced in size by 2-3 tracks and 2-3 columns.



**Figure 8.** Rather than expand cells to grid points as in the sidewalk approach, the flexible grid approach modifies the track and column structure of the channel. The channel is grid-based in the center, but the grid lines jog at the edges to meet up with non-gridded connections. (a) shows the standard orthogonal channel structure, and (b) shows a channel whose grid structure has been flexed. The flexible grid approach can result in tracks or columns that don't extend all the way across the channel.

The flexible grid approach distributes the sidewalks among the channels by jogging the track and column structure at the ends to match up with connection points that don't fall on grid lines. This is illustrated in Figure 8. In the flexible grid approach, wasted space occurs within the channel because some columns and channels cannot extend all the way across the channel. However, there appears to be less wasted space in this approach than in the sidewalk approach. In the worst case, the wasted space is equivalent to two tracks and two columns per channel. If connection points are sparse, however (and this is usually the case), the flexible grid approach has almost zero wasted space. We are still in the early stages of exploring this alternative.

## 10. User Interface

Magic displays the layout on a color display, and users invoke commands by pointing on the display with a mouse and then pushing mouse buttons or typing keyboard commands. Magic provides multiple overlapping windows on the color display. Each window is a separate rectangular view on a layout. Different windows may refer to different portions of a single cell, or to totally different cells. Windows allow designers to see an overall view of the chip while zooming in on one or more pieces of the chip; this permits precise alignments of large objects. Information can be copied from one window to another.

## 11. Technology Independence

Although Magic contains a considerable amount of knowledge about integrated circuits, the information is not embedded directly in code. All the circuit information is contained in a technology file that Magic reads. This file defines the abstract layers for a particular technology, the corner-stitched planes used to represent them, and the assignment of abstract layers to planes. It tells how to display the various layers and defines the semantics of the paint and erase operations from Section 5 (for example "if poly-metal-contact is painted over diffusion, erase the diffusion and place poly-metal-contact tiles on both the poly-diff and metal planes"). The technology file contains the design rules used in design-rule checking and in plowing. Lastly, it tells how to fill in the structural details of transistors and contacts when generating CIF for circuit fabrication. The technology file format is general enough to handle a variety of nMOS and CMOS processes. Our technology file for an nMOS process with buried contacts and single-level metal contains about 130 lines.

## 12. Implementation

The implementation of Magic was begun in February of 1983. By early April 1983, a primitive version of the system was operational. Although the first system was based on corner stitching and abstract layers, it provided user features only equivalent to Caesar. During the summer of 1983 implementa-

Subsystem	Implementation Status
Edge-based DRC	Operational 9/1/83
Hierarchical and Continuous DRC	Operational 11/1/83
Circuit Extraction	Not begun
Plowing	Simplified version operational 10/1/83 Full version expected 1/1/84
Net List Editing	Operational 5/1/83
Channel Decomposition	Expected 1/1/84
Global Router	Expected 2/1/84
Channel Router with Obstacle Avoidance	Operational 10/1/83
Multiple Windows	Operational 11/1/83

**Table II.** The implementation status of Magic.

tion was begun on the subsystems for routing, multiple windows, plowing, and design-rule checking. As of this writing, most of the advanced features are either operational or expected to be operational in the near future. See Table II. The system has been in use since April 1983 by the designers of a 32-bit microprocessor [8], and since September 1983 by several dozen students in an introductory VLSI design class.

Operation	Speed
Painting tiles into corner-stitched database	200 tiles/sec.
Design-rule checking	200 tiles/sec.
Simplified Plowing	100 tiles/sec.
Channel routing ("Deutsch's difficult example," 60 nets)	3 sec.

**Table III.** Some sample measurements of the speed of the Magic system. All measurements were made on a VAX-11/780.

Magic is written in C under the Berkeley 4.2 Unix operating system for VAX processors. The current implementation works only with AED color displays with special Berkeley microcode extensions. Altogether, Magic contains approximately 45000 lines of code. Table III gives a few sample performance measurements of pieces of the system.

### 13. Conclusions

We have not yet had enough designer experience with Magic to evaluate the system thoroughly, but the initial response has been favorable. The only major problem encountered so far has been one of education: if designers are accustomed to working with actual mask layers, then the abstract layers in Magic are confusing at first. This problem was exacerbated in the early versions of the system because the design-rule checker wasn't implemented. With continuous feedback from the checker, we hope that it will be much easier for designers to learn the abstract layers. We expect that the abstract layers will be easier for designers to work with than the actual mask layers, since they hide many irrelevant details.

The pieces of the Magic system work well together. Corner stitching appears to be a complete success: it provides all the operations needed to implement Magic's advanced features, and results in simple and fast algorithms. The design-rule checker's edge-based rule set meshes well with the



corner-stitched data, and is used also for plowing. The abstract layers simplify the design rules, provide information needed for plowing and circuit extraction, and simplify the designer's view of the layout.

We hope that Magic's flexibility will change the VLSI layout process in two ways. First, we hope that it will enable designers to experiment much more than previously. At the cell level, they can use plowing to rearrange cells quickly and easily. Cells can be designed loosely, then compacted. At the chip level, plowing and the routing tools can be used together to rearrange the floorplan, route the connections, compact or stretch, and try again. The ability to experiment means that students will be able to develop better intuitions about how to design chips; it also means that designers will be able to fix bugs and enhance performance more easily.

Second, we hope that Magic will make it easier to reuse pieces of designs. To design a new chip, a designer will select cells from a large library, use plowing and painting to make slight modifications in their shape or function to suit the new application, and perhaps design a few new cells. Then the routing tools will be used to interconnect the cells. We hope that this approach will result in a substantial reduction in design time for large circuits.

#### 14. Acknowledgements

As tool builders, we depend on the Berkeley design community to try out our new programs, tell us what's wrong with them, and be patient while we fix the problems. Without their suggestions, it would be extremely difficult to develop useful programs. The SOAR design team, and Joan Pendleton in particular, have been invaluable in helping us to tune Magic. Randy Katz, Dave Patterson, and Carlo Séquin all provided helpful comments on this paper. The Magic work was supported in part by the Defense Advanced Research Projects Agency (DoD) under contract N00034-K-0251, and in part by the Semiconductor Research Cooperative under grant number SRC-82-11-008.

#### 15. References

- [1] Chen, N.P., Hsu, C.P., and Kuh, E.S. "The Berkeley Building-Block Layout System for VLSI Design." Memorandum No. UCB/ERL M83/10, Electronics Research Laboratory, University of California, Berkeley, February, 1983.
- [2] Hamachi, G.T. and Ousterhout, J.K. "A Switchbox Router with Obstacle Avoidance." In this technical report.
- [3] Keller, K.H. and Newton, A.R. "KIC2: A Low-Cost, Interactive Editor for Integrated Circuit Design." *Proc. Spring COMPCON*, 1982, pp. 305-306.

- [4] Lee, C. Y. "An Algorithm for Path Connections and Its Applications." *IRE Transactions on Electronic Computers*, September 1961, pp. 346-365.
- [5] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI Layouts." *VLSI Design*, Vol. II, No. 4, Fourth Quarter 1981, pp. 34-38.
- [6] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." Technical Report UCB/CSD 82/114, Computer Science Division, University of California, Berkeley, December 1982. To appear in *IEEE Transactions on CAD/ICAS*, January 1984.
- [7] Ousterhout, J.K. "The User Interface and Implementation of Caesar." Technical Report UCB/CSD 83/131, Computer Science Division, University of California, Berkeley, August 1983.
- [8] Patterson, D.A. ed. "Smalltalk on a RISC." Final reports from CS292R, Computer Science Division, University of California, Berkeley, April 1983.
- [9] Rivest, R.L. and Fiduccia, C.M. "A Greedy Channel Router." *Proc. 19th Design Automation Conference*, 1982, pp. 418-424.
- [10] Rosenberg, J. et al. "A Vertically Integrated VLSI Design Environment." *Proc. 20th Design Automation Conference*, 1983, pp. 31-36.
- [11] Scott, W.S. and Ousterhout, J.K. "Plowing: Interactive Stretching and Compaction in Magic." In this technical report.

- [12] Taylor, G.S. and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." In this technical report.
- [13] Weste, N. "Virtual Grid Symbolic Layout." *Proc. 18th Design Automation Conference*, 1981, pp. 225-233.
- [14] Williams, J. "STICKS - A Graphical Compiler for High Level LSI Design." *Proc. 1978 National Computer Conference*, pp. 289-295.

# Magic's Incremental Design-Rule Checker

*George S. Taylor and John K. Ousterhout*

Computer Science Division  
Electrical Engineering and Computer Sciences Department  
University of California  
Berkeley, California 94720

## *ABSTRACT*

The Magic VLSI layout editor contains an incremental design-rule checker. When the circuit is changed, only the modified areas are rechecked. The checker runs continuously in background to keep information about design-rule violations up-to-date. This paper describes the basic rule checker, which operates on edges in the layout, and the techniques used to perform incremental checking on hierarchical designs.

**Keywords and Phrases:** design-rule checking, interactive layout editor

## 1. Introduction

Almost all existing design-rule checking (DRC) programs are batch oriented [1] [2]. They read in a complete circuit layout and check the entire design. If the circuit is changed, the only way to find out whether design rules have been violated is to recheck the entire design, no matter how small the change or how large the design. For chips with tens of thousands of transistors, batch DRC run may require hours of computer time.

This paper describes a different approach to design-rule checking. As part of the Magic VLSI layout editor [3], we have built a checker that operates *incrementally*. When the layout is modified, Magic records which areas have changed and rechecks only those areas. While the user continues editing, the checker runs in background and highlights errors as it finds them. There is no set-up time because it works from the same data structure used to represent the layout. Since most changes made with the interactive editor are small and the checker is fast, it can usually display errors instantly.

The user's view of design-rule checking is a simple one. As he edits the circuit, small white dots appear over areas that contain layout errors. As soon as the errors are fixed, the white dots go away. Error information is stored with the design and it will reappear during the next editing session if the violation has not been fixed. This information is always kept up-to-date, so there is never any need to run a batch checker.

In the next section, we describe Magic's internal representation for a layout and explain how particular features contribute to fast incremental checking. Section 3 describes how the basic checker works from edges in the layout and how design rules are specified. Section 4 shows how we use the basic checker for incremental checking of individual cells, and section 5 describes how hierarchical designs are handled. Section 6 gives measurements of the checker's speed.

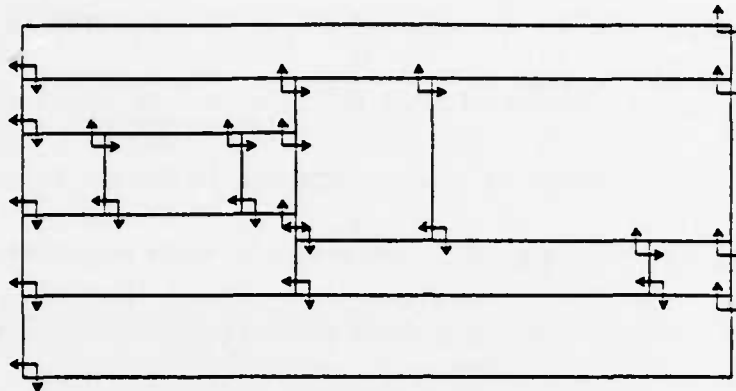
## 2. Representation of a Layout

In Magic, a layout is represented as a hierarchical collection of cells. Each cell contains mask information plus pointers to subcells. For now, we will consider only a single cell at a time (Section 5 generalizes the solution to handle hierarchical designs).

Magic represents the mask layers of a cell with rectangular *tiles*, which means that it handles only Manhattan geometries. Each tile indicates the type of mask layer it represents. Tiles are connected to form *planes* by a technique called *corner-stitching* [2] illustrated in Figure 1. The tiles in a plane are non-overlapping and cover it completely. Empty areas are covered with tiles of type "space."

Each cell contains several planes of mask information. Mask types that interact (such as polysilicon and diffusion) are stored together in the same





**Figure 1.** An example of a corner-stitched plane. Each plane contains tiles of different types that cover the entire area of the plane (space tiles are used where there is no mask material). Each tile contains four pointers that link it to neighboring tiles at its corners. The pointers make it easy to find all the material in a given area.

plane, while those that do not interact (such as polysilicon and metal) are stored in different planes. Contacts between mask types on different planes are represented in both of them. Our nMOS process has two planes: one for metal and one for polysilicon, diffusion, and transistors.

Instead of working directly with physical mask layers, Magic uses *abstract layers* to represent structures such as transistors and contacts. The abstract layers appear in the database as tiles with special types. For example, instead of representing an enhancement transistor as a polysilicon tile over a diffusion tile, it is represented with a tile of type "enhancement transistor." A more complete explanation of the abstract layers is given in [3]. What matters here is that all the interesting features are represented explicitly: there is no need to cross-register diffusion and polysilicon to discover the transistors.

The design-rule checker takes advantage of Magic's database in three ways. First, the corner-stitched tiles allow DRC to find material in a given area very quickly. Second, division of mask information into planes allows the checker to work with one plane at a time, ignoring irrelevant geometry on other planes. Third, there is no need to extract features by registering layers: the abstract layers represent the important features explicitly. Because of these features, there is no need for the checker to manage a separate structure of its own: it works directly from the layout database.

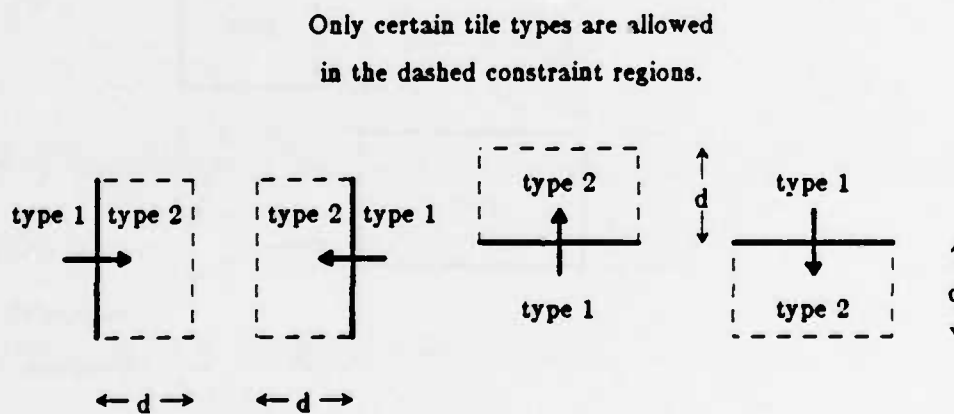
### **3. The Basic Checker**

This section describes the basic design-rule checking paradigm used to validate an area of a single corner-stitched plane. Later sections show how this basic checker is used to perform incremental checks on a single cell, and then on a hierarchy of cells.

#### **3.1. Edge-based Rules**

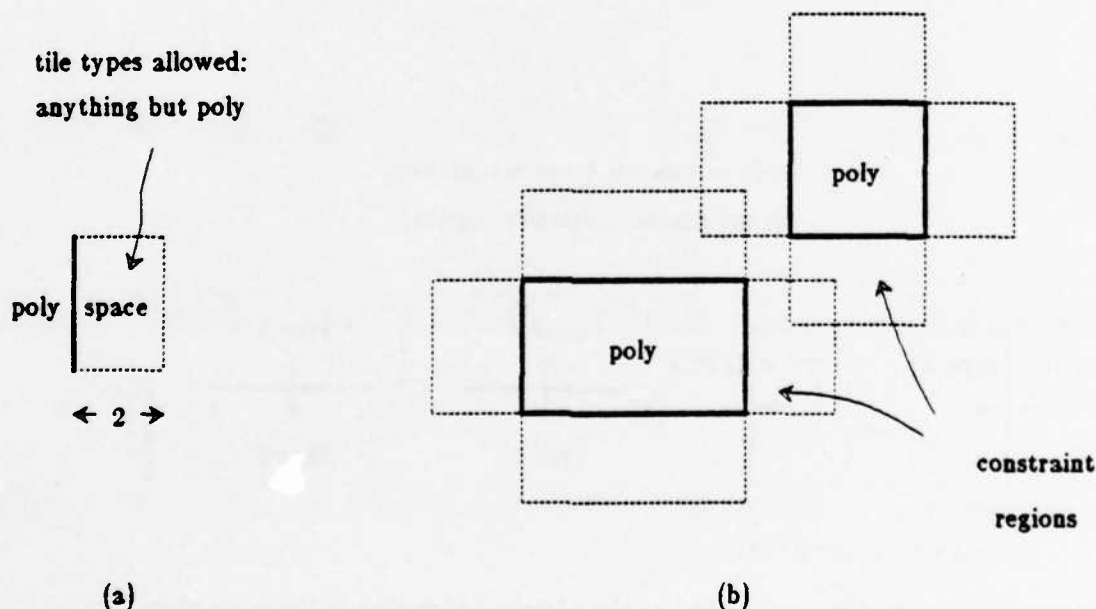
Magic's design rules are based on edges between tiles. Each rule can be applied in any of four directions, two for horizontal edges and two for vertical edges. The rule database contains a separate list of rules for each possible combination of materials on the first and second sides of an edge. In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on the second side of the edge. This

area is referred to as the *constraint region*. See Figure 2.

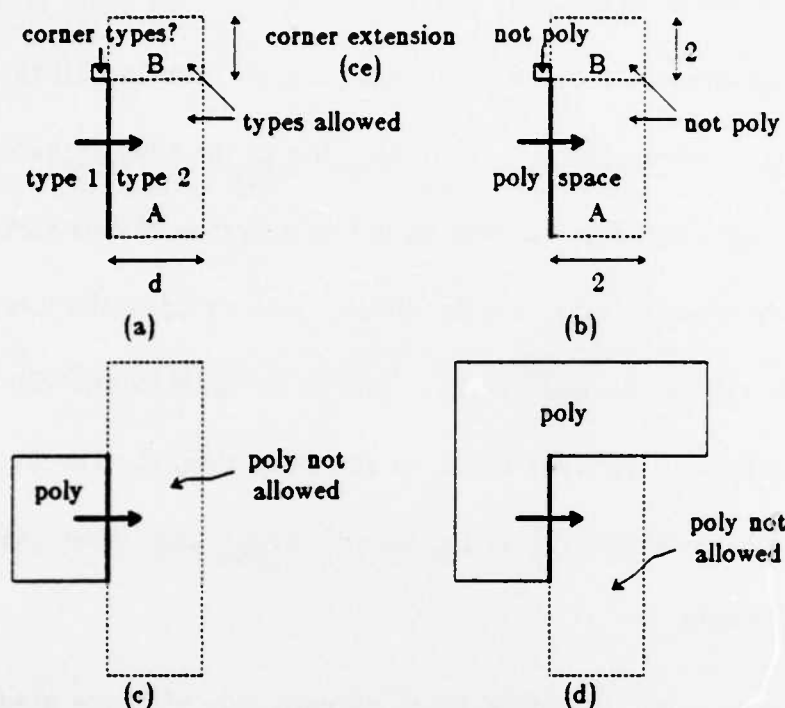


**Figure 2.** Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of *type 1* and *type 2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance *d* on *type 2*'s side of the edge.

Unfortunately, this simple scheme will miss errors in corner regions, as shown in Figure 3. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 4 for an illustration of the corner rules and how they work. Table 1 gives a complete summary of the information in each design rule.



**Figure 3.** If only the simple rules from Figure 2 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).



**Figure 4.** The complete design rule format is illustrated in (a). Whenever an edge has *type 1* on its left side and *type 2* on its right side, the area *A* is checked to be sure that only *types allowed* are present. If the material just above and to the left of the edge is one of *corner types*, then area *B* is also checked to be sure that it contains only *types allowed*. A similar corner check is made at the bottom of the edge. Figure (b) shows one of the polysilicon spacing rules, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge.

Parameter	Meaning
<i>type 1</i>	Material on first side of edge.
<i>type 2</i>	Material on second side of edge.
<i>d</i>	Distance to check on second side of edge.
<i>layers allowed</i>	List of layers that are permitted within <i>d</i> units on second side of edge.
<i>corner types</i>	List of layers that cause corner extension.
<i>ce</i>	Amount to extend constraint area when corner types match.

**Table 1.** The parts of an edge-based rule.

### 3.2. Applying the Rules

To check an area of a single plane, Magic must first find all the edges in that area. This is accomplished by searching for all the tiles in the area. The corner-stitched data structure is well suited to searches of this sort: see [4]. For each tile, the checker examines its left and bottom sides (the top and right sides of the tile will be checked by the neighbors on those sides). Since the tile may have neighbors of different types on the same side, the checker searches through all the neighbors to divide the side of the tile into edges with a single material on each side.

To process an edge, the mask types on each side of it are used to index into the rule table to find the list of rules for that kind of edge. Each rule in the list is checked, and white dots are displayed for any areas where the constraints are not satisfied. For each edge there are two rule applications: left-to-right and right-to-left (for vertical edges) or bottom-to-top and top-to-bottom (for horizontal edges). A different list of rules is applied in each direction, since the layers are reversed.

### 3.3. Specifying Design Rules

Design rules are specified in a technology file that contains the rules and other technology-specific information. When Magic starts executing, it reads this file and builds the rule table. Initially we specified rules in the detailed form of Table I, with one line for each edge rule. This scheme proved to be

unworkable, because there were many rules and it became difficult to convince ourselves that the rule set was complete and correct.

In order to simplify the process of creating rule sets, Magic now permits rules to be specified with high level macros for width and spacing. For example, the macro

**spacing ef DP 1**

is expanded into several rules to verify that types e and f (enhancement and depletion transistors) are always separated from types D and P (diffusion-metal contacts and poly-metal contacts) by at least one unit. The macro

**width pPBef 2**

is expanded into the set of edge rules needed to verify that the entire region containing any of the five types P, B, e, f or p (polysilicon) is always at least two units wide.

Most of the rules for our processes are simple width and spacing checks, so these two macros considerably simplify the writing of rule sets. Our nMOS rule set contains 8 width rules, 6 spacing rules, and 9 of the detailed edge rules for situations that cannot be handled by the width and spacing rules (e.g. transistor overhangs). Magic expands these 23 high-level rules into 126 detailed edge rules. The complete high-level rule set for nMOS is given in the Appendix.



The width and spacing macros make Magic's checker more efficient because the width and spacing rules are symmetric. If layers x and y are too close together, the violation can be detected from either an edge of x or an edge of y. This means that it is unnecessary to check the rules from both edges. Magic takes advantage of this symmetry by checking width and spacing rules in only two directions (left-to-right and bottom-to-top). In addition, symmetric rules mean that corner extension is only necessary on one end of each edge. Since most of the detailed edge rules come from the width and spacing macros, this speeds up the checking process by almost a factor of two.

#### **4. Continuous Design-Rule Checking**

This section shows how the basic checker is used to provide continuous incremental rule validation. As in the previous section, we consider only single-cell designs here.

In order to perform DRC incrementally, Magic maintains two extra kinds of information with each cell, stored in the same form as mask layers. First, Magic keeps information about rule violations that have been detected but haven't been corrected. The violations are represented by error tiles that cover the areas where rule constraints are not satisfied. The second kind of information consists of tiles describing the areas of the circuit that need to be reverified. The error tiles and the reverify tiles are stored in separate corner-

stitched planes. Each cell contains its own error and reverify planes.

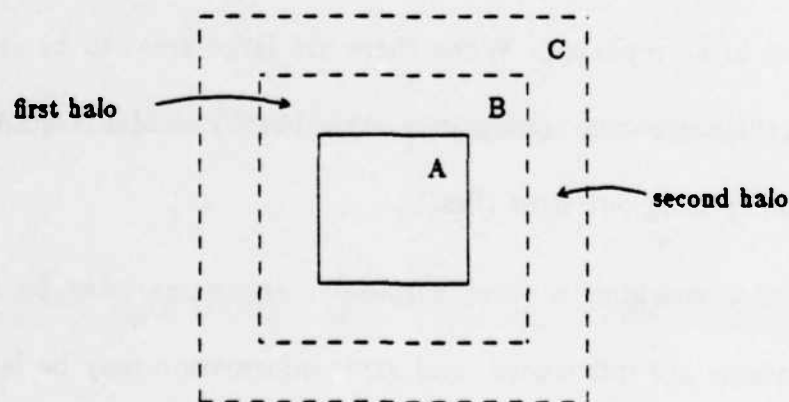
When a designer changes a cell, Magic creates reverify tiles that cover the area modified. The design-rule checker runs in background while Magic is waiting for the designer to enter the next command. DRC first searches for reverify tiles. Then it invokes the basic checker over the area covered by each tile found. The basic checker reverifies the area on each of the cell's planes, updates error tiles, and erases the reverify tile. Changes to the error information are reflected immediately on the graphics screen.

If the designer invokes a command while the checker is running, the checker stops so that the command can be processed without delay. After the command finishes, the checker resumes by starting over on the area that it was working on just before the interruption. Large reverify tiles are broken up into small ones before checking, in order to reduce the amount of work that might have to be repeated. When there are large areas to be reverified, the checker works across the design in a style like "Pac-Man," gobbling up reverify tiles and spitting out error tiles.

If incremental checking is done carelessly, errors may not be detected when new violations are introduced, and error information may be left in the database even after the violations have been corrected. Figure 5 illustrates the problem and Magic's solution. When an area is modified, error information may be affected in both the area that was modified and in the surrounding

area (for example, material in area A may be too close to something in the surrounding area B). We call the surrounding area the *halo*. Its width is equal to the largest distance in any design rule. Error information must be recomputed in the modified area and its halo. However, errors in the halo don't necessarily involve the inner modified area. They may come from interactions between the halo and a second halo outside it. To regenerate errors in the first halo correctly, information in the second halo must be considered.

If area A of Figure 5 were modified, Magic would recheck it by deleting all error information in A and B. The checker would then generate new error information in both areas by invoking the basic checker over areas A, B and C. Any errors found during this process would be clipped to the area of A and B, so that error information outside the region where errors were erased would not be affected.



**Figure 5.** If area A is modified, the design-rule checker erases existing error information in both A and B. Errors in B could have come from information in A, B or C, so all three areas must be checked to regenerate all of the errors. The width of the *halos* B and C is equal to the largest distance in any design rule.

The reverify and error tiles are stored with cells so that they are not lost at the end of an editing session. Normally, there will be no reverify tiles left at the end of a session, but if a large area has been changed recently, it is possible that it won't have been reverified when the session ends. In this case, the reverify tiles are written to disk with the cell. When the cell is read in during the next editing session, the design-rule checker will notice the reverify tiles and continue the reverification process. The reverify and error tiles are identical to the tiles used to represent mask layers, except that they are not manipulated directly by the designer.

## 5. Hierarchical Checking

Most of the layouts created with Magic consist of hierarchical cell structures rather than single cells (Figure 6). Each cell may contain subcells, and the subcells may overlap other subcells or mask information in the parent. A subcell may appear any number of times in any number of parents.

In hierarchical designs, errors can arise in any of three ways:

- a) the mask information of an individual cell may be incorrect;
- b) a subcell may interact incorrectly with another subcell; and
- c) a subcell may interact incorrectly with mask information in its parents.

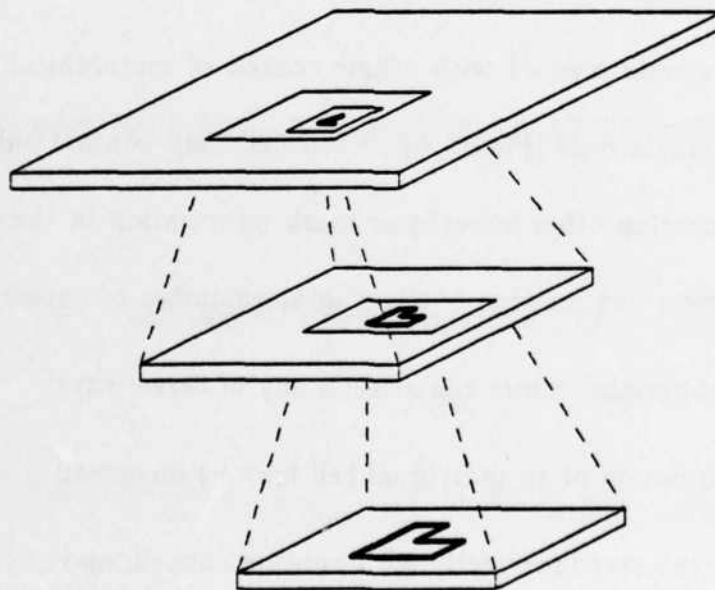
Magic's incremental checker includes facilities to detect all of these errors.

Overlapping subcells are no more difficult to handle than subcells that merely

about, because interaction errors are possible in either case.

### 5.1. Simple Checks and Interaction Checks

Two overall rules guide the hierarchical checker. First, the mask information in every cell is required to satisfy the design rules by itself, without consideration of subcells. Second, each cell and its subcells must together satisfy all the design rules, without consideration of how that cell is used in its parents. If the layout is viewed as a tree structure, the first rule means that each node of the tree must be consistent, and the second rule means that each subtree must be consistent.



**Figure 6.** Circuits are defined by cells arranged in a hierarchy. If mask information is changed in a low-level cell, Magic checks to be sure that the cell is consistent by itself and that there are no illegal interactions in parents or grandparents.

The overall rules result in two kinds of design-rule checking. The first rule is verified by running the basic checker over the planes containing mask information for each cell; this is called a *simple check*. The second rule is verified with an *interaction check* which considers interactions involving subcells. Each cell uses separate planes to hold its mask information, so interaction checks must combine information from different planes.

To make an interaction check on an area, the hierarchical structure is "flattened" to produce a new set of corner-stitched planes that combines all the information from all cells in the area to be checked. This includes mask information from the parent cell, plus mask information from subcells and sub-subcells, and so on. Once all the mask information in the area has been collected into a single set of planes, the basic checker is invoked on these planes in the standard fashion (halo expansion is performed as described in Section 4). Errors arising from the interaction check are placed in the parent cell.

Interaction checks are more expensive than basic checks, since they involve flattening a piece of the hierarchy. Fortunately, interaction checks can often be avoided. For example, if an area contains no subcells, then there is no need to perform an interaction check on that area. A simple check will find all errors. The interaction check can also be avoided if there is only a single subcell in an area, with no other subcells or mask information nearby. In

this case any errors must come from within the subcell, and those errors will be found by checks made within that cell. Interaction checks are necessary only in areas where a subcell is within one halo distance of mask information or another subcell. Even then, we only need to check the area around the interaction.

## 5.2. Checking Upward in the Hierarchy

When a cell is modified, simple checks and interaction checks have to be performed within that cell, and also within its parents in the hierarchy. For example, suppose mask information has been edited within a cell. Then a simple check must be performed within that cell, as well as an interaction check if there are subcells near the modified area. However, these two checks are not sufficient. If the modified cell is a subcell of other higher-level cells, then the change may have introduced interaction problems within the higher-level cells. For each parent of the modified cell, an interaction check must be performed over the area of the modification. Interaction checks must also be performed in grandparents, and so-on up to the top-level cell in the hierarchy. In the cell that was modified, both simple and interaction checks must be performed, but in the parents and grandparents only interaction checks are necessary.

Magic uses two kinds of verify tiles to handle the two kinds of checks. When a cell is modified, "verify-all" tiles are placed in that cell to signify that both simple and interaction checks must be performed. At the same time,

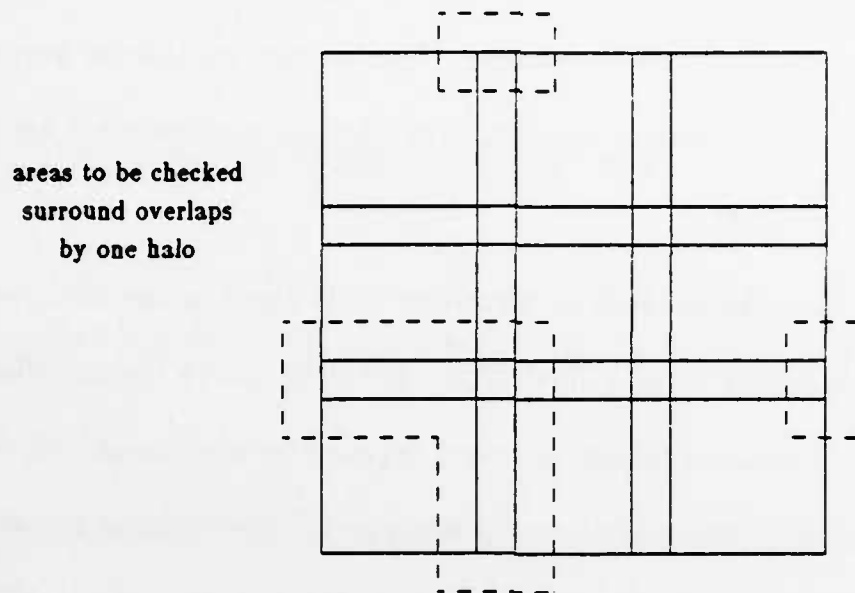


"verify-interactions" tiles are placed in parents and grandparents to indicate that interaction checks have to be performed. The background checker keeps track of which cells in the database contain verify tiles and performs each kind of check wherever necessary.

In the worst case, the hierarchical algorithm could result in the modified area being rechecked once at each level of the hierarchy above the cell that was changed, with a separate flatten operation required for each check. However, in deep hierarchies most of the interaction checks are avoidable: in cells far above the modified one, the modified area will almost certainly appear in the middle of a single subcell with no mask information or other subcells nearby. Unless there are many large subcell overlaps, any given area of mask information is likely to require an interaction check at only one point in the hierarchy.

### 5.3. Arrays

One other form of hierarchical check arises because Magic has an array construct. To simplify the creation of cell arrays, Magic contains a special array facility: each subcell may consist of either a single instance or a one- or two-dimensional array of identical instances. Because of the array construct, there is actually a third overall rule that guides the hierarchical checker: each array must satisfy all the design rules, independently of other information in the parent containing the array. Whenever a change is made to an array, the



**Figure 7.** An array is internally consistent if the three dotted areas satisfy the design rules. All possible interactions between elements of the array are identical to the ones that occur these three regions.

array structure is reverified by checking the three areas shown in Figure 7.

## 6. Implementation and Performance

The design-rule checker is written in C. Its 2000 lines of code are divided into roughly equal thirds for building the internal rule table from the technology file, implementing the basic checker on one plane, and providing for hierarchical checking.

The incremental checking system has just recently become operational. We've made preliminary measurements on single cells with the untuned system. The basic checker processes 200 tiles per second on a VAX 11/780 running Unix. To compare Magic's performance with that of other systems, we

state their speeds in terms of transistors checked per second in Table 2.

A typical change to a circuit involves only a few tiles, so the cost of incremental reverification is dominated by the size of the halos. From this, we estimate that roughly 50 tiles have to be checked per command in an nMOS design. This requires about one-fourth of a second of CPU time.

The average number of edges found per tile is 2.5, but only 1.8 of these have different mask types on the two sides of the edge. An average of 1.7 rules are applied per non-trivial edge.

## 7. Conclusions

Magic's design-rule checker demonstrates that incremental checking is feasible. We think that circuit designers will find that continuous feedback reduces the time needed to create new designs or modify existing ones. The key to the incremental checker is low overhead: the ability to run from the same database as the interactive editor, the ability to find important edges in the layout quickly, and the ability to find nearby material quickly. The two

System	Transistors / second
Lyra [2]	2
Baker [1]	3
Mart [5]	6-8
Magic	10-15

**Table 2.** Performance of several design rule checkers. All of the programs were run on a VAX 11/780.

features of Magic's database that reduce overhead are the corner-stitched tile planes and the abstract mask layers. Extending the checker to work in hierarchical designs frees the designer from tedious reverification of interactions when subcells are revised.

## 8. Acknowledgements

Gordon Hamachi, Bob Mayo, and Walter Scott all participated in discussions that led to the incremental checker and provided many useful comments on drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), under Contract No. N00034-K-0251.

## 9. References

- [1] C. M. Baker and C. Terman, "Tools for verifying integrated circuit designs," *Lambda* (now *VLSI Design*) Vol. 1, No. 3 (1980), pp. 22-30.
- [2] M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 19th Design Automation Conference*, June, 1982, pp. 530-36.
- [3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "Magic: A VLSI Layout System," included in this technical report.

- [4] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," Technical Report UCB/CSD 82/114, Computer Science Division, University of California, Berkeley, December, 1982. To appear in *IEEE Transactions on CAD/ICAS*, January, 1984.
- [5] B. J. Nelson and M. A. Shand, "An Integrated, Technology Independent, High Performance Artwork Analyzer for VLSI Circuit Design," Technical Report VLSI-TR-83-4-1, VLSI Program, Division of Computing Research, CSIRO, Eastwood, SA 5063, Australia, April, 1983.

## 10. Appendix

To illustrate how Magic is programmed for a particular technology, this section lists the design rules for an nMOS process with buried contacts and a single level of metal. Most rules are specified using width and spacing macros which Magic expands into detailed lower-level rules. Detailed edge and four-way rules may also be specified directly. Table 3 gives the abbreviations that we use for the names of mask types.

Poly/Diffusion plane:	s	space
	d	diffusion
	p	polysilicon
	D	diffusion-metal contact
	P	polysilicon-metal contact
	B	buried contact
	e	enhancement transistor
	f	depletion transistor
Metal plane:	s	space
	m	metal
	X	metal-diffusion contact
	Y	metal-polysilicon contact

**Table 3.** Single letter abbreviations for the names of mask types.

The rules in Table 4a define minimum line widths and feature sizes. The first three rules are for the line widths of diffusion, metal and polysilicon. The last five rules define the sizes of contacts and transistors. The *types* field may include one or more mask types. Magic creates a detailed edge rule for all combinations of one member of the *types* field, and one of the mask types in the same plane that is not included in the *types* field.

	types	d	reason
width	dDBef	2	diffusion
width	pPBef	2	polysilicon
width	mXY	3	metal
width	D	4	diff/metal contact
width	P	4	poly/metal contact
width	B	2	buried contact
width	e	2	efet
width	f	2	dfet

Table 4a. Width rules.

Table 4b contains spacing rules. We distinguish between spacing rules for types that can never be adjacent and spacing rules that apply only when two pieces of material are separated. In either case, Magic creates a number of detailed edge rules in a manner similar to that for width rules.

The width and spacing macros can be used to specify most symmetrical constraints for a particular technology. The detailed edge rules created from the width and spacing macros are applied only from left-to-right across

	types 1	types 2	d	can be adjacent?	reason
spacing	ef	DP	1	no	transistor - contact
spacing	e	f	3	no	efet - dfet
spacing	B	e	3	no	buried contact - efet
spacing	dDBef	dDBef	3	yes	diff - diff
spacing	pPBef	pPBef	2	yes	poly - poly
spacing	mXY	mXY	3	yes	metal - metal

Table 4b. Spacing rules.



vertical edges in the layout, and from bottom-to-top across horizontal edges.

These edge rules always check one corner, also.

To specify asymmetrical constraints and constraints that apply alongside edges but not in corners, we use the explicit edge and fourway rules listed in Table 4c. The fourway rules are applied in both directions across all edges in the layout. They also trigger corner checks on both ends of every edge. The edge rules in Table 4c are similar to the ones derived from the width and spacing macros, but could not be written conveniently in either of those forms.

	type 1	type 2	d	layers allowed	corner types	ce	reason
edge	d	spP	1	s	spP	1	diff - poly spacing
edge	p	sdD	1	s	sdD	1	diff - poly spacing
edge	D	sp	1	s	sp	1	diff - poly spacing
edge	P	sd	1	s	sd	1	diff - poly spacing
fourway	ef	s	1	0	0	0	trans can't touch space
fourway	B	dD	4	sdpDPBef	sdpDPBef	3	b,c - cft spacing
fourway	f	B	3	B	0	0	b,c next to dft must be 3sf
fourway	ef	p	2	p?	p	2	poly overhang transistor
fourway	ef	d	2	dD	d	2	diff overhang transistor

Table 4c. Edge and fourway rules.

# Plowing: Interactive Stretching and Compaction in Magic

*Walter S. Scott and John K. Ousterhout*

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California  
Berkeley, CA 94720

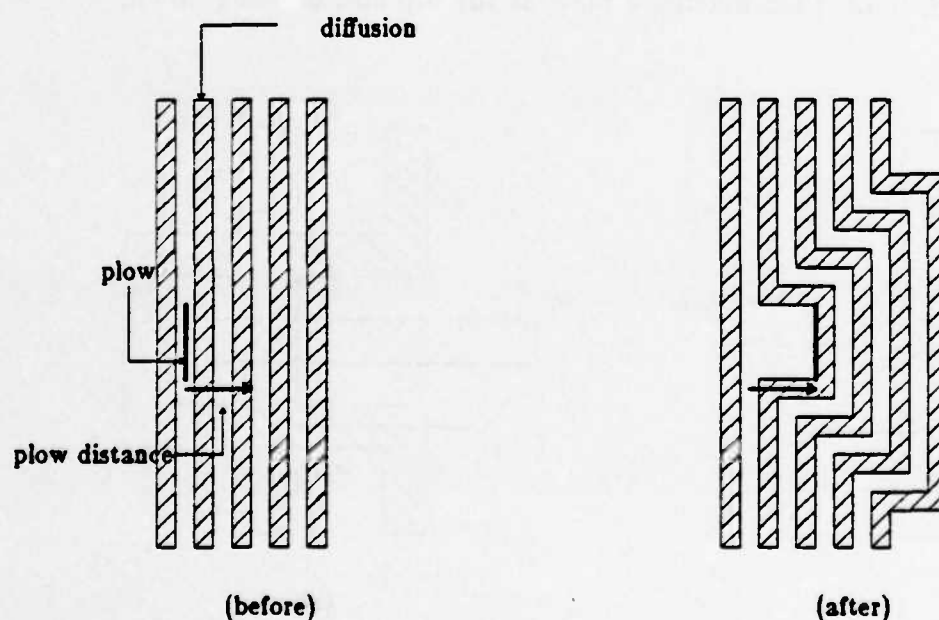
## Abstract

The Magic layout editor provides a new operation called *plowing*, for stretching and compacting Manhattan VLSI layouts. Plowing works directly on the mask-level representation of a layout, allowing portions of it to be rearranged while preserving connectivity and layout-rule correctness. The layout and connectivity rules are read from a file, so plowing is technology independent. Plowing is fast enough to be used interactively. This paper presents the plowing operation and the algorithm used to implement it.

**Keywords and Phrases:** interactive layout editor, stretching, compaction.

## 1. Introduction

Plowing is a new operation provided by the Magic layout editor [OHMST 84] for stretching and compacting Manhattan VLSI layouts. It allows designers to make topological changes to a layout while maintaining connectivity and layout rule correctness. Plowing can be used to rearrange the geometry of a subcell, compact a sparse layout, or open up new space in a dense layout. In a hierarchical environment plowing also allows cell placement to be modified incrementally without the need for rerouting. To avoid dependence on a particular technology, plowing is parameterized by a set of layout and connectivity rules contained in a technology file.



**Figure 1.** Plowing opens up new space in a dense layout. Geometry is pushed in front of the plow, subject to layout-rule constraints. The connectivity of the original layout is maintained. Jogs are inserted automatically where necessary.

Conceptually the plowing operation is very simple. The user places either a vertical or a horizontal line segment (the *plow*) over some part of a mask-level representation of the layout, and then gives the direction and the distance the plow is to move. Plowing can be done up, down, to the left, or to the right. (The rest of this paper will assume plowing to the right.) The plow is then moved through the layout by the distance specified. It catches vertical edges (boundaries between materials) as it moves and carries them along with it. Since only edges are moved, material behind the plow is stretched and material in front of the plow is compressed. Figure 1 shows how plowing can be used to open up new space. Figure 2 shows how it can be used for stretching. Plowing can be used to compact an entire cell by placing a plow to the left and plowing right, then placing a plow at the top and plowing down.

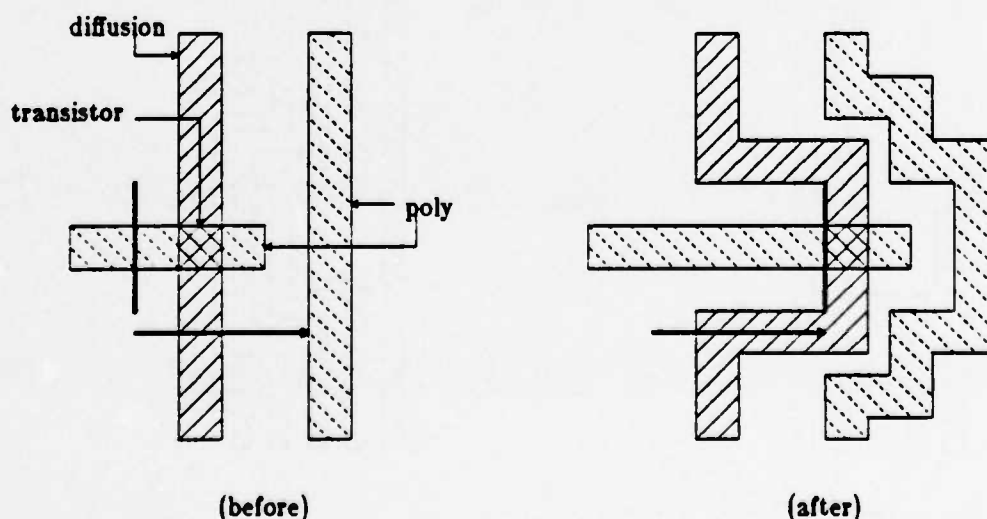


Figure 2. Material to the left of the plow is stretched. Material to the right is compressed. Objects such as transistors do not change in size.

Plowing is so named because each of the edges caught by the plow can cause edges in front of it to move in order to maintain connectivity and layout-rule correctness. These edges can cause still others to be moved out of the way, recursively, until no further edges need be moved. A mound of edges thus builds up in front of the plow in much the same manner as snow builds up on the blade of a snowplow.

Section 2 of this paper discusses plowing in the context of previous work. Sections 3 and 4 introduce the plowing algorithm for a single mask layer. Section 5 extends it to multiple mask layers and hierarchical designs. Finally, Section 6 presents performance measurements and our experience with plowing in the Magic system.

## 2. Background

VLSI layouts are difficult to modify. Because of this, designers are often committed to the initial choice of implementation, rather than being able to experiment with alternatives. Existing cells often cannot be re-used in subsequent designs because they don't quite fit; it is typically easier to redesign a new cell from scratch than to modify an old one. Bugs in a dense layout are hard to fix, leading to a debugging cycle which can take days.

Many of these difficulties stem from the fact that seemingly small changes to a layout can have disproportionately large effects. Sometimes this is for

electrical reasons. For example, in ratio logic such as nMOS, changes in the size of one transistor may necessitate changes in the sizes of others. However, even purely topological changes—those which preserve the electrical properties of the layout—can require much more work than the size of the change would suggest. As Figure 1 illustrated, merely opening up new space in a layout can cause effects which ripple outward over a much larger area. Rearranging the internal geometry of a cell or modifying the placement of cells in a floor plan can be similarly expensive because of the need to maintain connectivity with the surrounding material.

Previous attempts to cope with the re-arrangement problem have used symbolic design or sticks [RBDD 83, West 81, Will 78]. In the symbolic/sticks approach, designers enter layouts in an abstract form containing zero-width wires, contacts, and transistors. The sticks form is then run through a compactor to generate actual mask information. As part of the compaction, the circuit elements are moved as close together as the layout rules permit. In a sticks design style, cells can be designed loosely without worrying about exact spacings, since the spacings will be determined by the compactor. However, it is not necessarily easy to make major changes to a sticks cell once it has been entered. Virtual grid systems like Mulga and VTVID provide mechanisms for adding new grid lines uniformly across a cell, but it is still difficult to make large topological changes.

The plowing approach has all the advantages of sticks. It allows cells to be designed loosely and then compacted. In addition, plowing can be used to rearrange cells or open up new space, either across the whole cell or in one small portion. Small changes can be made in one area without having to recompact the entire cell (a global recompactation may potentially shift every geometry in the cell). The plowing approach lets the designer see the final sizes and locations of all objects as he is editing; in the sticks approach, it is hard to predict the final structure of a cell from its abstract form, so compaction must be used frequently to see the results of a change to the sticks.

### 3. Simple plowing algorithm

Plowing works by finding *edges* and moving them. An edge is a boundary, parallel to the plow, between material of two different types. When an edge moves, the material to its left is stretched, and the material to its right is compressed. In this section we will describe how plowing works when only a single mask layer is present. This material will be assumed to have a minimum width of  $w$ , and a minimum separation of  $s$ . Edges will always be boundaries between this material and "empty" space.

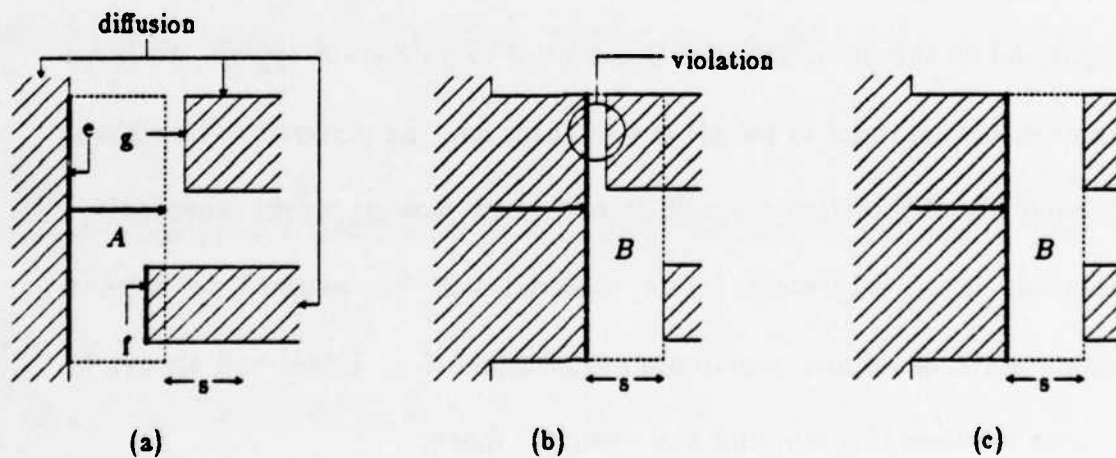
The fundamental step in plowing is to move a single edge. This step involves determining which other edges must move as a consequence of this motion. The following discussion presents plowing as though it moves a given



edge by first recursively sweeping all other edges out of its way, and then sliding the edge into the newly opened space. Section 4 will present a better scheme for ordering edge motions than this depth-first recursion.

### 3.1. Finding edges

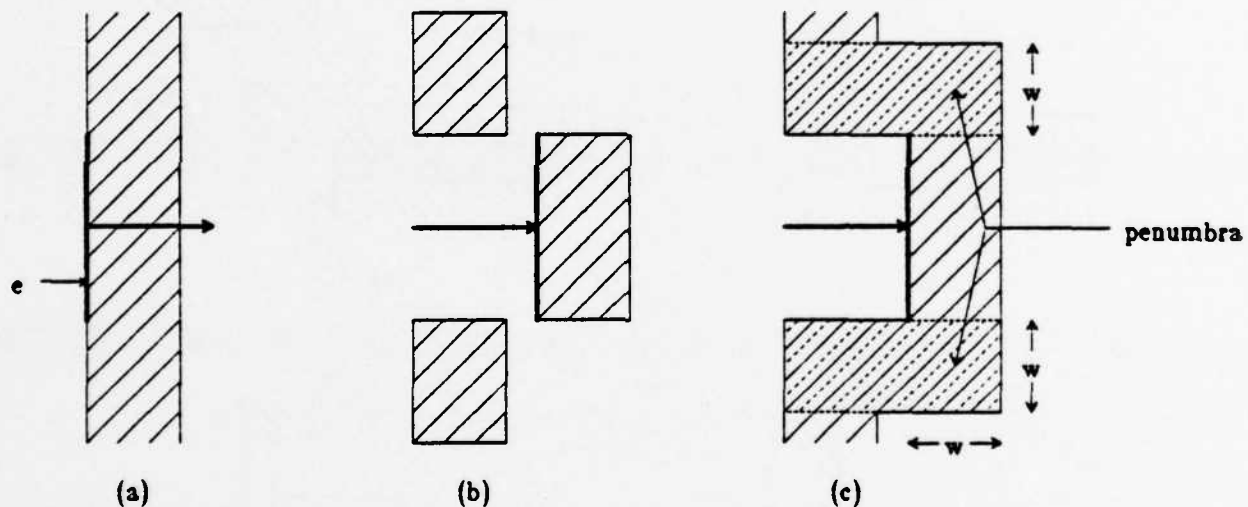
Figure 3 depicts a trivial layout consisting of three unconnected pieces of diffusion. The edge labelled  $e$  is to be moved to a final position indicated by the arrowhead. This could be either because  $e$  was caught by the plow, or because it is being moved to make room for some edge to its left. At a very minimum, the rectangular area labelled  $A$  must be swept clear of any material before the edge can be moved. However, because of the spacing rule, any material inside area  $B$  would then be too close to the newly moved edge. Con-



**Figure 3.** When the edge  $e$  moves, all edges in area  $A$  (the area swept out by  $e$ ) must be moved (a). Moving only these edges results in edge  $f$  moving but not edge  $g$ . This leaves a layout-rule violation (b) between  $e$  and  $g$ . Searching area  $B$  as well as area  $A$  avoids this problem. The two areas are referred to collectively as the *umbra* of edge  $e$ .

sequently, the area to be swept includes both areas  $A$  and  $B$ . The union of these two areas is referred to as the *umbra* of the edge  $e^*$ .

Plowing must also search above and below the umbra to prevent the edge from sliding too close to other edges above or below it. Figure 4a shows why this is necessary. If material were moved out of the umbra alone, as in Figure 4b, the result is electrical disconnection. To avoid this, plowing must also move edges out of the areas above and below the umbra. The correct result is

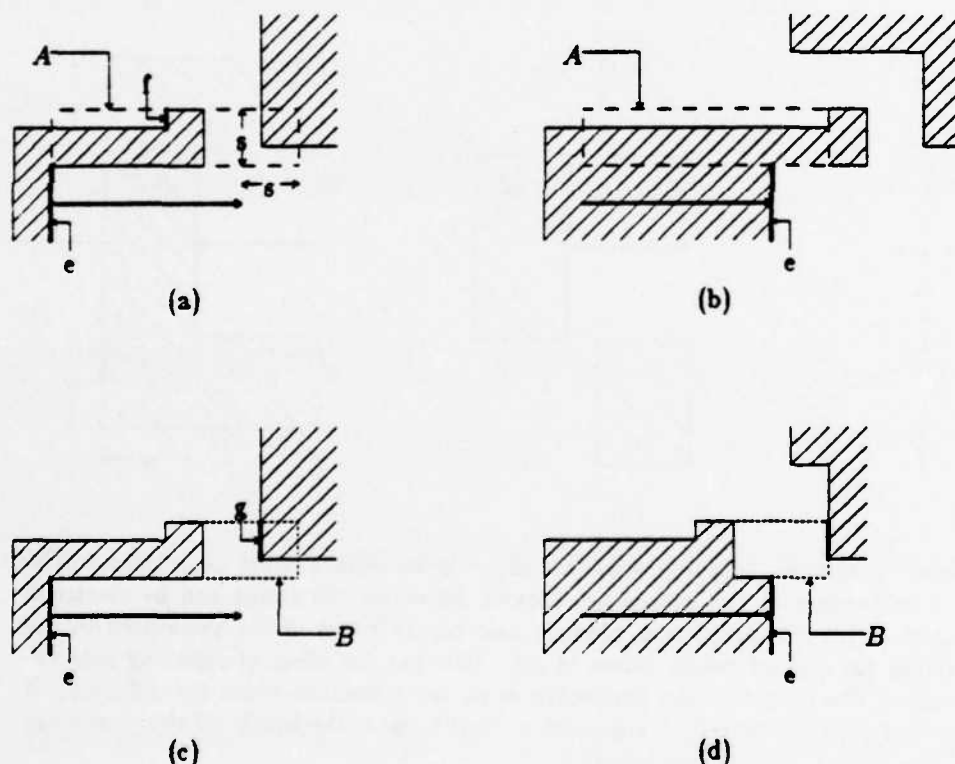


**Figure 4.** When the edge  $e$  moves (a), edges in its umbra must be moved to the right. If only edges in the umbra are moved, however, the result can be electrical disconnection (b). To avoid this, plowing also moves edges in the *penumbra* to the right, giving the correct result shown in (c). This has the effect of inserting jogs automatically. The height of the penumbra is  $w$ , the minimum-width for diffusion. If diffusion had been to the left of  $e$  instead of to the right, the height of the penumbra would have been  $s$ , minimum-separation.

\* In a solar eclipse, the *umbra* is that portion of the moon's shadow from which the sun appears to be completely eclipsed. The *penumbra* is the part of the shadow surrounding the umbra from which the sun appears only partially eclipsed. In plowing, the umbra contains edges directly in the path of an edge being moved, while the penumbra contains edges not in the path but nonetheless too close.

shown in Figure 4c. The areas above and below the umbra are referred to collectively as the *penumbra*. Jog insertion is an automatic consequence of searching the penumbra. Moving edges out of the penumbra also prevents electrical shorts, as can be seen by reversing the roles of material and space in Figures 4a-4c.

The left-hand boundary of the penumbra is not always aligned with the edge being moved. Instead, this boundary is formed by following the outline

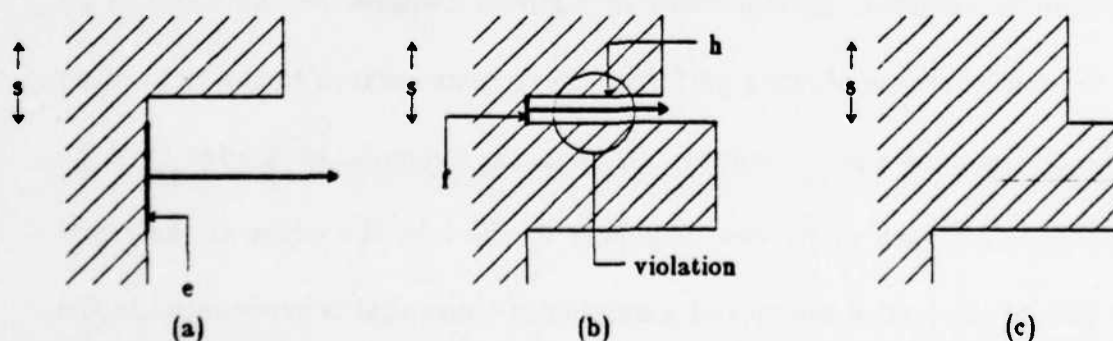


**Figure 5.** If *e*'s penumbra included all of area *A*, as shown in (a), then edge *f* would be found and moved, resulting in (b). This is undesirable, since *f* need not move in order to preserve layout-rule correctness and connectivity. A better definition of the penumbra would be area *B* only, as shown in (c). Searching this area would result in only the edge *g* being found and moved, as is necessary to preserve layout rule correctness.

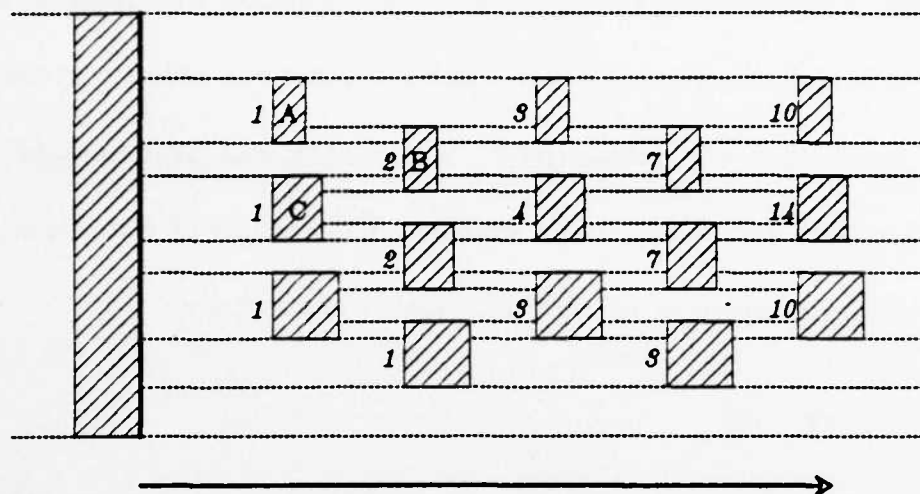
of the material forming the edge, as illustrated in Figure 5. This insures that the penumbra contains only those edges which must move in order to preserve layout rule correctness and connectivity. The umbra and penumbra of an edge are collectively referred to as its *shadow*. The shadow of  $e$  contains all the edges which must move as a direct consequence of moving  $e$ .

### 3.2. Sliver prevention

The rules described in Section 3.1 guarantee that plowing never moves one vertical edge too close to another. However, they do allow violations to be introduced between horizontal segments that are formed when material is stretched. These violations take the form of slivers of material or space whose height is less than the minimum allowed. Eliminating such slivers requires that their left-hand edges be moved, as illustrated in Figure 6. The left-hand edge of each sliver lies along the left-hand boundary of the penumbra, so it can be found when tracing the outline of the penumbra.



**Figure 6.** When the edge  $e$  moves (a), a sliver of space is introduced below the horizontal segment  $h$ , as shown in (b). To correct this, the left-hand edge of this sliver,  $f$ , is moved along with  $e$ , but only as far as the right-hand end of the segment  $h$  (c).



**Figure 7.** This lattice structure causes exponential worst-case behavior in the depth-first plowing algorithm when edges in the shadow are processed from top to bottom. The objects (A, B, etc.) must be incompressible to cause this worst-case behavior. Object B is moved once when object A moves, then slightly farther when object C moves. The numbers to the left of each object show how many times each of its edges is moved.

#### 4. Breadth-first vs. Depth-first Search

In the previous section, plowing was described as a depth-first search in which all edges to the right of a given edge were moved before the edge itself. While this approach is conceptually clear, it has poor worst-case behavior. An N-tier lattice structure as illustrated in Figure 7 requires on the order of  $2^N$  edge motions, because plowing performs the recursive search to the right of an edge each time the edge is moved. If, as in the example, each edge must be moved once for each of its two neighbors to the left, the edges at the right-hand side of the lattice are moved a number of times that is exponential in the number of tiers.

Instead, plowing waits until the final position of an edge is known before it performs the search to the right of that edge. This strategy causes the number of edge motions to be linear in the number of edges in the lattice. (A detailed explanation is given in [Oust 84].)

A simple way to insure that edges are moved only once their final positions are known is to use breadth-first search. Magic maintains a list of edges to be moved, sorted in order of increasing  $x$ -coordinate. On each iteration, the leftmost edge is removed from the list and the shadow to its right is searched. Any edges discovered by this search are placed in the list along with the amount they must move. Since the final position of an edge can only be affected by edges to its left, the final position of the leftmost edge in the list is always known.

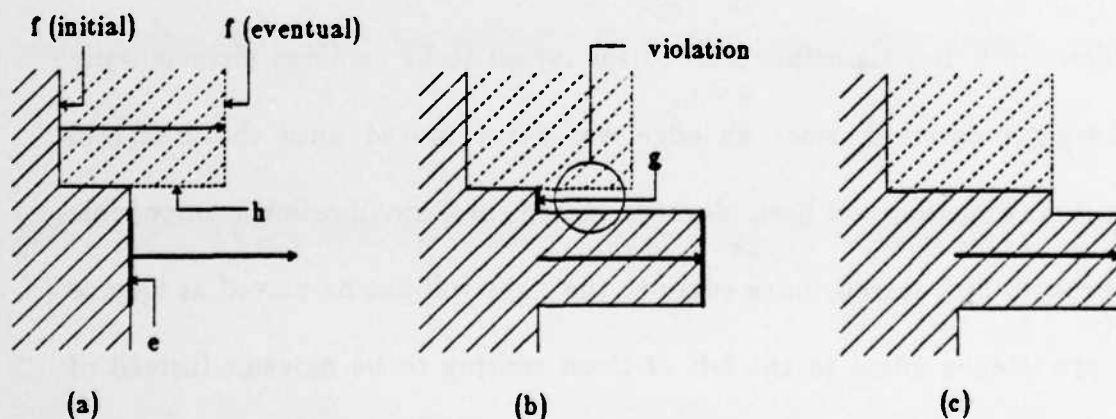
The depth-first algorithm allowed the layout to be modified incrementally as plowing progressed, since an edge was never moved until the area into which it was moving had been cleared. Incremental modification is impossible with breadth-first search, since edges to the right will not be moved as long as there are queued edges to the left of them waiting to be moved. Instead of actually updating the layout as it progresses, the breadth-first version of plowing stores with each vertical edge segment the distance it moves. When the shadows of all edges have been searched, and the distance each edge moves has been determined, plowing invokes a post-pass to update the layout from



the information stored with each edge.

However, if the layout is not modified until all edges have been processed, special care must be taken to avoid the generation of slivers. Figure 8 illustrates the problem. To process each edge correctly, it is important to know what other edges have been already been processed and what their final positions will be. In general, the plowing algorithm must consider edges whose final positions will be in the shadow, rather than those whose initial positions are in the shadow.

The success of the breadth-first algorithm depends on the fact that left-to-right plowing never changes the order of edges along any horizontal line, and never changes any vertical coordinates. Furthermore, edge has stored



**Figure 8.** When processing an edge in the breadth-first approach, it is important to use information about the final positions of edges that have already been processed. In (a), it has already been decided to move edge  $f$ , but the edge will not actually be moved until all other edges have been processed. If edge  $e$  is processed without considering the new position of  $f$ , a sliver will result as shown in (b). Instead, the plowing algorithm must consider the eventual positions of edges that have already been processed, to produce the result of (c).



with it the distance it is going to move. As a consequence, plowing can use the initial layout structure for searching, and yet can easily find all objects whose final coordinates fall in a given area.

## 5. Extensions for real layouts

This section extends the simple plowing algorithm of the previous two sections to handle multiple mask layers. Plowing is also extended to handle features, such as transistors and contacts, whose size should not be changed, and to allow noninteracting mask layers, such as metal and polysilicon, to slide past each other. Finally, since layouts in Magic may be hierarchical, this section closes with a description of how plowing handles hierarchy.

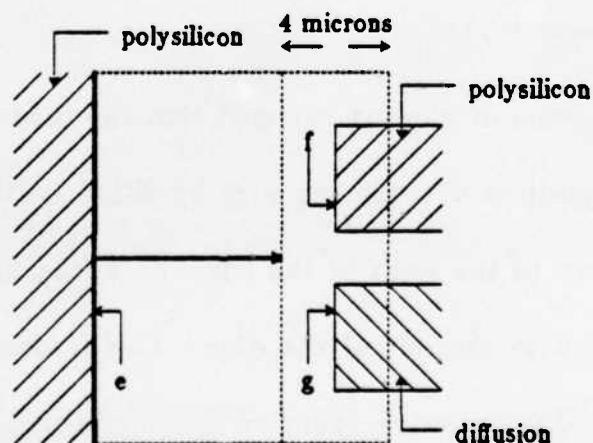
### 5.1. Multiple mask layers

The simple version of plowing assumed that the shadow extended to the right of the final position of a moving edge by either  $w$  (the minimum-width rule) if material lay to the right of the edge, or  $s$  (the minimum-separation rule) if material lay to the left of the edge. This insured that the shadow included all edges directly in the path of the edge being moved. Since the same layout rule applied between the edge being moved and any other edge, all edges found during the search of the shadow would have to move.

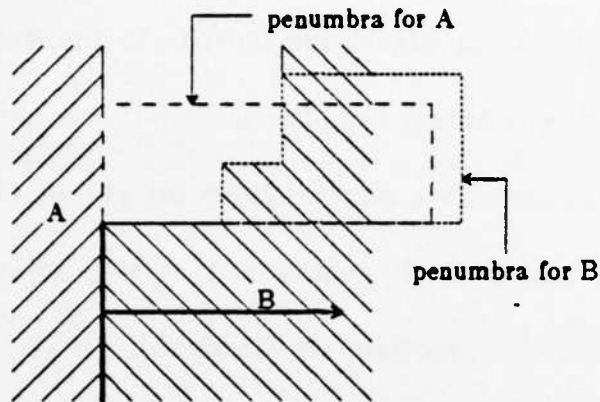
With more than one mask layer there may be more than one layout rule to apply for a given edge. For example, in our nMOS process, the minimum

separation between diffusion and polysilicon is 2 microns, while that between two pieces of diffusion is 6 microns. Both of these rules apply at an edge between diffusion and empty space.

To insure that the shadow contains all edges which must move, the shadow must extend beyond the area the edge sweeps out by the worst-case layout rule distance applying to that edge. As Figure 9 illustrates, however, not all of the edges found in the shadow search will actually need to move. Each edge found must be checked for its minimum allowable separation from the edge being moved. Fortunately, this can be done very quickly using the same techniques as those used in Magic's incremental layout-rule checker [TaOu 84].

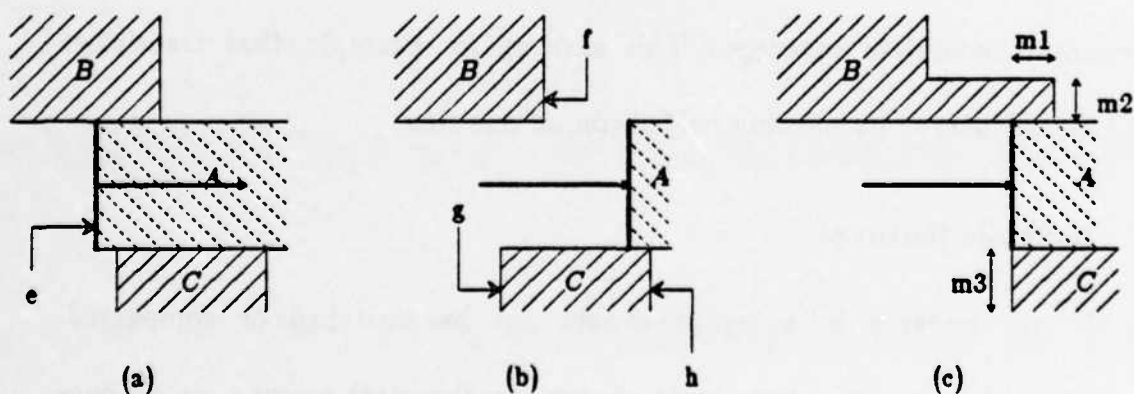


**Figure 9.** The area of a shadow search is determined by the worst-case layout rule. However, not all edges in that area will have to be moved. Edge *f* must move, because the separation between two polysilicon features must be 4 microns and edge *e* approaches to within 2 microns of *f*. Edge *g* need not move since the minimum separation between polysilicon and diffusion is only 2 microns.



**Figure 10.** An edge between two different types of material has a penumbra for each. The spacing rules for material of type A are applied in A's penumbra. The minimum-width rule for material of type B is applied in B's penumbra. The sizes of each penumbra may be different because of the different layout rules applied in each.

If the edge being moved has material on both sides, there is really a penumbra for each type of material. The layout rules applied while searching each penumbra will in general be different. Slivers must be prevented along the boundaries of both penumbra. See Figure 10 for an example.



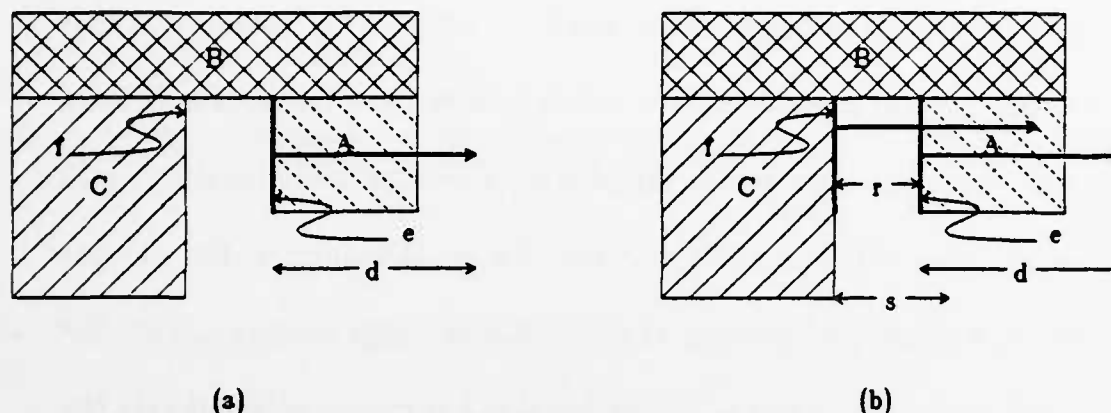
**Figure 11.** If edge *e* is plowed, material A may disconnect from B and C. To prevent this, a minimum-width segment of edges *f* and *g* is dragged along with *e*. The edge *g* is moved not to maintain connectivity (which would have been achieved by moving *h*), but to prevent C from being uncovered. In (c), *m1* is the lesser of the minimum widths for A and B, *m2* is the minimum width for B, and *m3* is the minimum width for C.

Multiple mask layers require extra caution to maintain connectivity with material above and below an edge being moved. In the single-layer scheme, the penumbra search guarantees that the material does not become disconnected. However, the penumbra search follows the outline of a single type of material, so it will not by itself guarantee that two adjacent materials of different types will remain connected (see Figure 11).

Special actions must be taken during the penumbra search to handle horizontal edges between different materials. First, if two materials share a horizontal edge, then Magic guarantees that one material does not slide past the end of the other: it maintains a minimum-width connection between the two (this is the case between materials A and B in Figure 11). Second, if one material completely covers the edge with another material (for example, the A-C edge in Figure 11), Magic plows the other material as much as is needed to maintain complete coverage. This ensures, for example, that transistors don't get uncovered by plowing polysilicon off one side.

## 5.2. Inelastic features

Certain features in a layout should not be stretched or compacted. Transistors, for example, have sizes chosen for electrical reasons, as do contacts. Our discussion of edge motion has assumed that the material forming both sides of the edge was stretchable. When material is inelastic, both its left-hand and right-hand edges must be moved in tandem. In particular, if the



**Figure 12.** When inelastic objects are present, plowing may have to cope with circular dependencies. Material *B* is inelastic, and *A* and *C* are both minimum-width. When edge *e* moves by distance *d* in (a), object *B* must move by the same distance to prevent *A* from being uncovered. To prevent *C* from being uncovered, *C*'s left-hand edge must move, finally causing edge *f* to move by distance *d*. Edge *e* is in *f*'s shadow as a result, but should not be moved a second time.

right-hand edge of a piece of inelastic material moves, its left-hand edge must move also.

A consequence of inelasticity is that moving an edge can cause motion of edges to its left, possibly resulting in a circular dependency. The example in Figure 12 illustrates such a dependency. The depth-first plowing algorithm is completely incapable of resolving such a dependency. The breadth-first algorithm resolves it by comparing the amount an edge is supposed to move with the motion distance already stored with the edge. If the stored motion distance is greater, the edge need not be moved a second time.

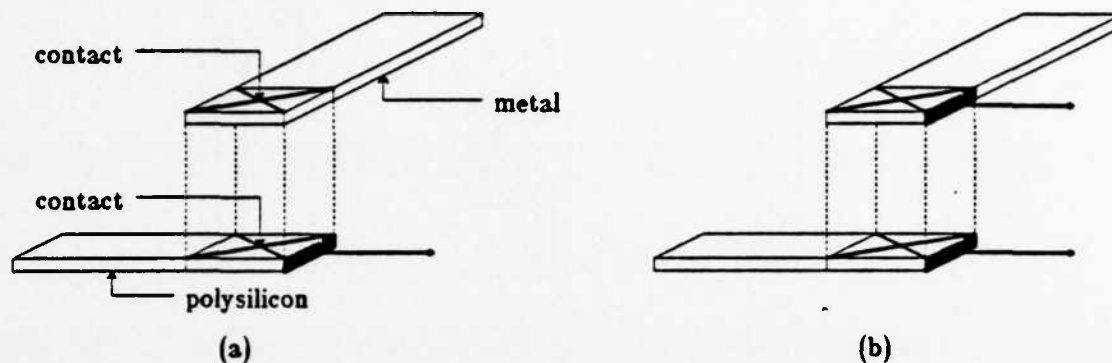
If the distance *d* between edges *f* and *e* in Figure 12 is less than *s*, the minimum separation allowed (ie, there is currently a layout rule violation), looking at the motion distance of *e* is insufficient. When the shadow of *f* is

searched, plowing is supposed to move all edges found far enough away so that they cause no rule violations with the newly moved  $f$ . This would mean that edge  $e$  would have to move by  $d+s-r$ , which is more than the motion distance stored with the edge. As a result, the plowing algorithm loops infinitely, each time moving edge  $e$  by an additional  $s-r$ . To avoid infinite walks, plowing never moves a shadowed edge (eg,  $e$ ) more than the edge causing the shadow (eg,  $f$ ). This technique prevents infinite looping, but preserves layout rule violations existing in the original layout.

### 5.3. Noninteracting planes

Section 4 explained that the order of vertical edges along a horizontal line is unchanged by plowing. Thus material being plowed can never slide over other material in its path. There are cases, however, where it is desirable that certain materials in a layout move independently. Metal, for example, does not interact with either polysilicon or diffusion except at contacts, so it should be able to slide over them.

To allow sliding, Magic segregates the mask information in a layout into a collection of non-interacting *planes*. Material in one plane is free to slide past material in any other plane. The nMOS technology, for example, has two planes: one to hold metal wires, and one to hold polysilicon, diffusion, and transistors.



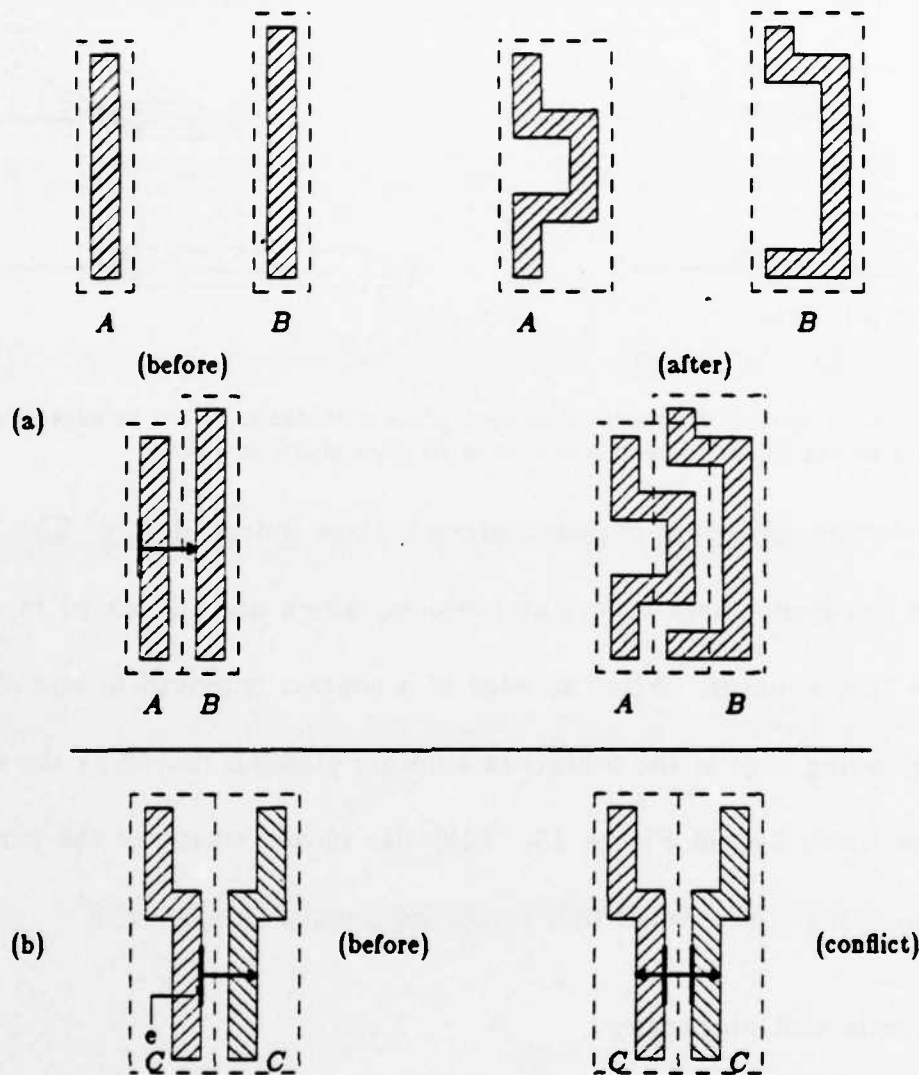
**Figure 13.** A contact is duplicated on each plane it connects. When an edge of a contact is moved on one plane, it is moved on all other planes as well.

The plowing algorithm operates on each plane independently. The only interaction between planes occurs at contacts, which are duplicated in each plane that they connect. When an edge of a contact is moved in one plane, the corresponding edge of the contact in all other planes is moved by the same amount, as illustrated in Figure 13. This also moves whatever the contact connects to in the other planes, thus preserving connectivity.

#### 5.4. Subcells and hierarchy

One approach for plowing a hierarchical layout, such as that shown in Figure 14a, is to treat it as though it were non-hierarchical and propagate edge motions inside subcells. This might be workable when no subcell is used more than once. However, Magic instantiates subcells by reference, so a change in one instance of a subcell is reflected in all its other uses. Situations in which a subcell is used more than once can produce unsatisfiable sets of constraints, as Figure 14b illustrates.





**Figure 14.** Plowing in the presence of hierarchy. (a) Plowing might treat hierarchy as though it were invisible to the user. Each of cells *A* and *B* would be modified. (b) Cell *C* is used twice, once flipped left-to-right and once in its normal orientation. Both uses refer to the same master definition of *C*. Moving edge *e* to the right is impossible, because it requires *e* to move to the left in order to keep out of its own path. The more edge *e* is moved to the right in the left-hand use, the worse the violation becomes.

Magic takes a simpler approach, which is to view subcells as black boxes to which connectivity must be maintained by plowing, but whose internal structure should not be modified. A consequence of Magic's approach is that

plowing can be used to modify the placement of cells at the floor plan of a chip, since it only changes the location of subcells, not their contents.

When any mask geometry that abuts or overlaps a cell is moved, the entire cell must move by the same amount. Conversely, whenever a subcell moves, all mask geometry and other subcells that abut or overlap it must also move by the same amount. The net effect is that a cell behaves like flypaper, causing all geometry over its area to "stick" to it and move as a whole when any part of it is required to move.

In addition to preserving connectivity with subcells, when plowing moves other geometry it must avoid introducing any layout rule violations with the geometry inside a subcell. One approach for dealing with this is to define a *protection frame* [Kell 82] for each cell, an outline around the cell into which no material may be plowed. Magic uses an extremely simple form of protection frame: it assumes that the cell contains all types of material right up to the border of its bounding box.

For example, in our nMOS rule set, the worst-case layout rule involving diffusion is the diffusion-diffusion spacing rule of 6 microns. An edge with diffusion to its left can be plowed to within 6 microns of a subcell before that subcell will itself have to move. The worst-case rule distance involving polysilicon is 8 microns, so polysilicon can only be plowed to within 8 microns of a subcell before the cell must move. Since the contents of subcells are con-

sidered unknown, the closest one subcell can be plowed to another before the other will have to move is the worst-case layout rule in the entire ruleset, which in our ruleset is 8 microns. Of course, if the user wishes to overlap two cells, he can still do that using other editing operations beside plowing.

## 6. Results and experience

Plowing has been implemented as part of the Magic VLSI layout system. It is written in C under the Berkeley 4.2 Unix operating system for VAXes. A simplified version of plowing (corresponding to that described in Sections 3 and 4) has been operational since October of 1983.

While the full implementation of plowing has not been completed, measurements on the simple version indicate that it is fast enough to be used interactively. An example similar to that presented in Figure 1a, consisting of 48 parallel bars of polysilicon each separated by 4 microns (the minimum separation), took 3.2 seconds of VAX-11/780 CPU time to produce a result similar to that in Figure 1b. Only 1.0 seconds were spent computing the edge motions; the remainder of the time was spent in the post-pass which actually updates the layout.

## 7. Acknowledgements

Gordon Hamachi, Robert N. Mayo, and George Taylor all contributed to the discussions out of which the plowing algorithm arose. In addition to the above people, Randy Katz, Ken Keller, and Steve and Jean McGrogan all provided helpful comments on early drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

## 8. References

- [RBDD 83] Rosenberg, J., Boyer, D., Dallen, J., Daniel, S., Poirier, C., Poulton, J., Rogers, D., Weste, N. "A Vertically Integrated VLSI Design Environment." *Proceedings, 20th Design Automation Conference*, 1983, pp. 31-38.
- [Kell 82] Keller, K., Newton, A. "A Symbolic Design System for Integrated Circuits." *Proceedings of the 19th Design Automation Conference*, June 1982.
- [Oust 81] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI." *VLSI Design*, Vol. II, No. 4, Fourth Quarter 1981, pp. 34-38.
- [Oust 84] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." To appear in *IEEE Transactions on CAD/ICAS*, Vol 3, No. 1, January 1984.
- [OHMST 84] Ousterhout, J.K., Hamachi, G., Mayo, R.N., Scott, W.S., and Taylor, G.S. "The Magic VLSI Layout System." In this technical report.
- [TaOu 84] Taylor, G.S., and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." In this technical report.

Plowing

December 2, 1983

[West 81]      Weste, Neil. "Virtual Grid Symbolic Layout." *Proceedings, 18th Design Automation Conference*, 1981, pp. 225-233.

[Will 78]      Williams, J. "STICKS- A Graphical Compiler for High Level LSI Design." *Proceedings of the 1978 NCC*, May 1978, pp. 289-295.

# A Switchbox Router with Obstacle Avoidance

*Gordon T. Hamachi  
John K. Ousterhout*

Computer Science Division  
Department of Electrical Engineering  
and Computer Sciences  
University of California  
Berkeley, California 94720  
(415) 642-9716, 642-0865

## ABSTRACT

This paper presents a new switchbox router developed as part of the Magic layout system. Based on Rivest and Fiduccia's "greedy" channel router, the Magic router is capable of routing channels containing obstacles such as preexisting wiring. It jogs nets around large obstacles and multi-layer obstacles such as contacts. Where unable to avoid large single-layer obstacles, it river-routes through them. It combines the effectiveness of traditional channel routers with the flexibility of net-at-a-time routers.

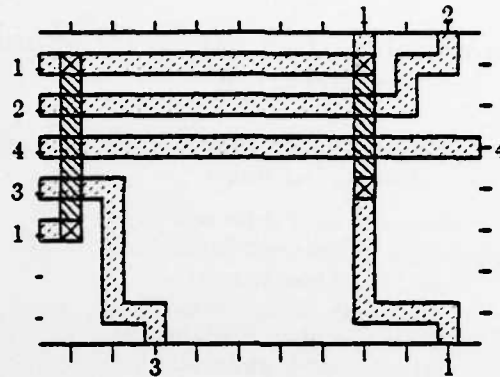
**Keywords and Phrases:** channel routing, physical design aids, layout, VLSI.

## 1. Introduction

Previously placed wires such as power and ground routing form obstacles in routing areas. We have developed a new switchbox router as part of the Magic layout system [OHM], capable of routing channels containing such obstacles. The router's novel aspect is its ability to both avoid obstacles and consider interactions between nets as channels are routed. It thus combines good features from net-at-a-time routers and traditional channel routers.

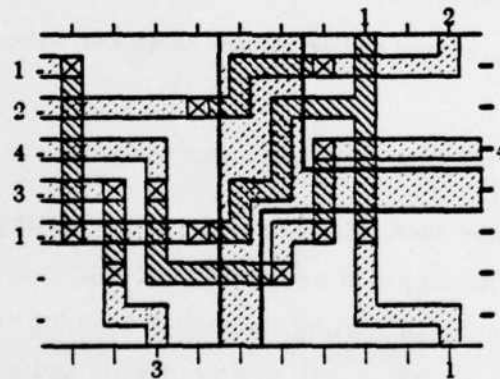
The Magic router is an extension of Rivest and Fiduccia's "greedy" channel router [RiF]. It performs a column by column scan of a rectangular routing region. At each column it applies a series of rules controlling the placement of vertical jogs within the column.

Figure 1 shows the solution to a simple routing problem. Figure 2 shows the same routing problem with an obstacle in the routing area. It illustrates some basic principles of obstacle avoidance. As the router extends nets from left to



**Figure 1.** A simple channel routing problem. Numbers around the border of the channel represent pins associated with signal nets. Pins with identical net numbers are connected by the router using a left to right, column by column scan.

right, it tries to avoid large obstacles in the columns ahead by jogging around them. If nets can not jog around large single layer obstacles they *river route* through the obstacles, switching layers if necessary.



**Figure 2.** The problem from Figure 1 with obstacles in the channel (drawn with heavy outlines). The router tries to cross obstacles at narrow points. If necessary it river-routes through obstacles.

Section 2 motivates the problem of obstacle avoidance and describes the Magic router's goals. Section 3 summarizes the "Greedy" router, upon which our work is based. Section 4 presents our solutions to a number of problems encountered in adapting the greedy channel router to avoid obstacles. In section 5 we present extensions for routing switchboxes. Section 6 provides a detailed view of the router. Section 7 describes a channel splitting mechanism. The paper



concludes with a discussion of the router's implementation and performance.

## 2. Motivation

Automated routing systems typically divide the routing of chips into three steps: *channel definition*, *global routing*, and *channel routing*. In the channel definition step, empty areas between cells are divided into non-overlapping rectangular *channels*. The global routing step selects the sequence of channels through which each signal net will be routed to make the desired connections. The channel routing step assigns physical locations to the wires in each channel, realizing the signal routings specified in the global routing step.

A standard model for channel routing assumes a grid of two independent layers of minimum width wiring. Horizontal *tracks* are wired in one of these layers, while vertical *columns* are wired in the other layer. Connections between layers are made with *contacts*; where no contacts appear, layers may cross over each other.

Routers generally assume that channels start off completely free of wiring. Thus, it is impossible to use an automatic router with pre-routed wires. This is a serious limitation since certain signals such as power, ground, and clock lines have special restrictions on width, layer, and length which existing channel routers fail to handle.

Because channel routers do not tolerate the presence of obstacles, designers must either accept inferior results generated by automatic routing systems or try to hand patch the router output. Hand patching is difficult because automatic routers leave little room to add wires or to move wires to different layers. Also, if the chip has to be rerouted, the hand patching must be completely redone.

Channel routing systems that do handle obstacles have done so in restricted ways. The PI system [Riv] has the notion of "covered channels" -- areas wired with metal for power and ground routing, through which other signals may be river routed in polysilicon. It fragments large channels containing metal power and ground wiring into many smaller covered and uncovered channels each of which must be routed individually. The problem of routing one large channel is thereby reduced to the problem of routing several smaller, more constrained

channels.

The BBL system [Che], [CHK] handles prewiring from a separate power and ground routing phase. It routes power and ground signals near the edges of channels. BBL then ignores the power and ground routing areas except to bridge other signals across them. It routes all other signals using only the clear parts of the channels. BBL does not allow any hand routing.

Routers using maze [Lee][Hig] routing methods are able to avoid obstacles on multiple layers. The problem with these routers is that they consider only one net at a time. Since they completely route a single net before considering the next net, they cannot consider interactions between nets as a channel is routed. For this reason these routers are inferior to true channel routers for channel routing of general layouts [Sou].

The Magic router provides a general obstacle avoidance capability that combines the advantages of the above approaches. It allows designers to prewire critical nets, putting them at any position and in any layer. The Magic router routes around these prewired nets.

The router considers interactions between nets. Routing decisions are based on an overall strategy rather than on a net-at-a-time basis. Considering tradeoffs between alternatives improves the overall quality of the resulting wiring.

Magic uses single-layer obstructed areas to do useful routing. Since large, loosely constrained areas are easier to route, it avoids fragmenting these obstructed areas into small, highly constrained, hard to route areas.

Particularly in interactive design environments nearly optimal results obtained quickly are more useful than optimal results obtained after long computation. Our router is fast, and produce good results.

### **3. The Greedy Router**

Since the greedy router algorithm is the starting point from which our algorithm was developed, we start with a brief overview of its operation. Three features of the greedy router are of particular importance. First, the greedy router makes a column by column scan of the routing area. It completely wires the current column before extending active tracks into the next column. Second,

it uses a list of rules to control the placement of vertical wiring in a column. Rules are applied in order of importance, to a) avoid "getting stuck"; and b) to make subsequent columns easier to route (Figure 3). The list of rules can easily be modified. Third, unlike constraint graph approaches, the greedy router allows *split nets*, nets that occupy more than one track at a time. Split nets give the router the flexibility to evaluate alternatives and choose the one that is best for the overall routing problem.

Column wiring begins by bringing the nets of a column's top and bottom pins (if any) into the first tracks that are either vacant or already assigned to the nets. Deferring this to a later step might allow vertical wiring to block a net, preventing it from being brought into a vacant track.

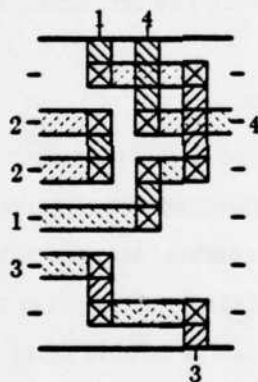


Figure 3. Three columns wired by the greedy router. In the first column net 2 makes a collapsing jog and net 3 makes a falling jog. In the second column net 4 enters the channel, preventing net 1 from making a collapsing jog; however, net 1's lower track makes a jog to reduce the range of tracks assigned to this split net.

Bringing a net into the first available track may leave it *split* on multiple tracks. Split nets can fill up the channel, making it impossible to bring in additional nets. The greedy router thus makes collapsing split nets its next priority. Since conflicting vertical wiring can make it impossible to collapse all split nets in a particular column, the router collapses split nets in the pattern that frees up the most empty tracks for use in the next column.

Vertical wiring conflicts may prevent the router from collapsing all split nets. The router simplifies the routing of these remaining split nets by reducing the range of tracks occupied by these nets. It jogs each split net's highest occupied

track downward and its lowest occupied track upwards. The remaining problem is easier because collapsing can be done with shorter jogs.

Next, unsplit *rising* and *falling* nets are jogged upward or downward toward the edge of the channel with their next pin. This step anticipates the split nets that will be created when upcoming pins' nets are brought into the channel. It attempts to reduce the range of these split nets before they are created. This step prevents split nets if the rising or falling net can be jogged into what would otherwise be the first vacant track seen by a net as it enters the channel from a top or bottom pin.

The handling of split nets and rising and falling nets are examples of decisions based on interactions between nets. Among conflicting alternatives (a jog to raise a rising net may block a jog to lower a falling net) the router chooses the one that does the best job of simplifying the remaining overall problem.

#### 4. Extending the Greedy Router

In modifying the greedy router to avoid obstacles we had to solve a number of problems. The result was an augmented set of rules for placing horizontal and vertical wiring. In the following discussion, an area with a single layer obstacle is called an *obstructed* area. The Magic router river-routes through obstructed areas. An area is *blocked* if it contains a double layer obstacle. No routing may pass through blocked areas.

As it scans a channel from left to right, the greedy router expects that it can always extend a track into the next column if necessary. The router must avoid extending tracks into *blocked* areas (Figure 4).

We solve this problem by anticipating upcoming obstacles and attempting to jog nets out of their way. We do this by identifying areas near obstacles; these areas are called *obstacle thresholds*. A preprocessing step searches the routing area, marking obstacle thresholds. Tracks extending into these marked areas make *vacating* jogs to tracks outside these areas.

Another important issue is the tradeoff between horizontal and vertical wiring. Magic has to decide whether to route horizontal wires or vertical wires over single layer obstacles. It can not do both of these, since an obstacle and a wire

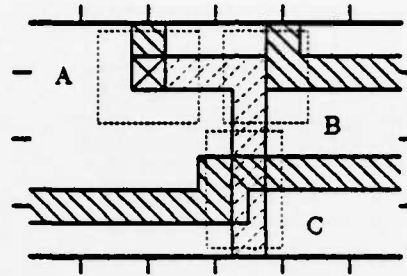


Figure 4. Tracks can not extend into blocked areas (drawn in dotted lines). Note that two adjacent areas of different layers (B) form blocks because there is no place to put contacts to bridge from one area to the other.

crossing it block both routing layers. A thin vertical wire should be bridged horizontally by tracks. Likewise, a thin horizontal wire should be bridged vertically by columns. Intermediate cases are harder to classify (Figure 5).

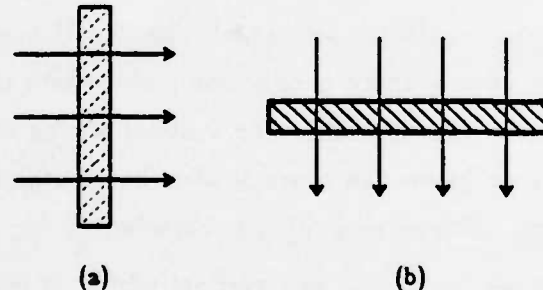
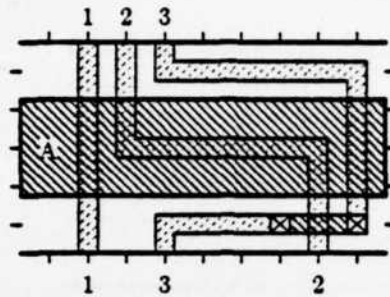


Figure 5. Thin width vertical wires should be bridged horizontally by tracks (a). Thin width horizontal wires should be bridged vertically by columns (b). Intermediate cases are harder to classify.

We solve this problem by always giving priority to horizontal wiring. If vertical wiring is not done in the current column it may be done in some later column. Horizontal wiring is more important: if the router needs to extend tracks but can not, it fails.

Although horizontal wiring gets priority over vertical wiring, we attempt to avoid extending tracks into large single layer obstacles. When tracks do extend into single layer obstacles the Magic router tries to jog them out of these areas, into unobstructed tracks. It is important to do this because a single track running through an obstructed area blocks all columns that might cross the obstructed area (Figure 6).



**Figure 6.** Nets avoid obstructed tracks wherever possible. Failure to do so may create blocked areas. Since net 2 is in an obstructed area, net 3 is forced to make a long detour.

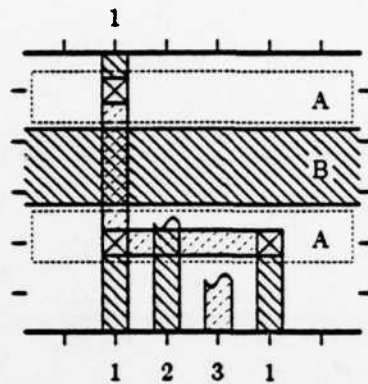
The greedy router assumes that it can make vertical column wiring anywhere the channel is not blocked by vertical wiring it previously placed. The Magic router has to know not only when to place vertical column wiring, but also how to do this. It has to know when areas are blocked, and when to place contacts to switch layers.

Given our wiring model, contact placement is simple. If a contact needs to be placed to allow a layer switch, there is only one place where that contact can go: immediately adjacent to the obstacle. For vertical wiring contacts may be placed immediately above or below the obstacle. For horizontal wiring the locations are immediately to the left and right of the obstacle.

Our wiring model allows horizontal and vertical wiring in either layer; however, only one layer of horizontal wiring and one layer of vertical wiring is allowed at any point. There is a preferred layer in each direction; horizontal tracks and vertical column wires may run in the opposite layer only to bridge an obstacle. Since poly is the preferred vertical layer, a vertical run may bridge a metal obstacle without placing contacts, but contacts need to be placed to bridge a poly obstacle. If the track immediately above the poly obstacle is vacant, then the contact can be placed. If the track is occupied by horizontal wiring, the preferred layer policy says that it must be in metal. The metal/poly boundary blocks the vertical run, since there is no space to bridge the metal track in poly and place a contact before running over the poly obstacle (Figure 7).

The greedy router assumes that channels can be arbitrarily expanded and that terminals on the left and right edges of the channel can "float" up and down





**Figure 7.** The outlined areas (A) above and below the obstacle (B) are reserved for column contacts necessary if the obstacle is to be bridged vertically. The router tries to keep the areas clear of wiring. Note that the horizontal metal run prevents both the poly (2) and the metal (3) vertical runs from bridging the obstacle, because there is no room to place contacts.

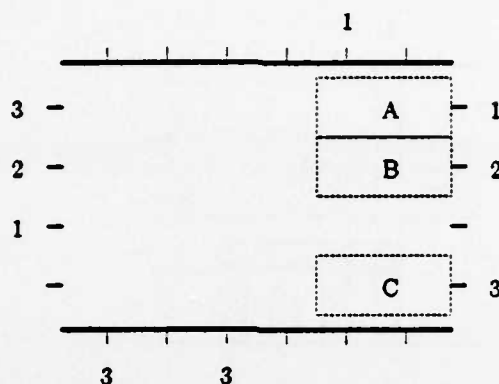
as long as their relative positions remain the same. Tracks may be inserted whenever the router gets "stuck". The Magic router assumes that channels have a fixed number of tracks and that terminals have fixed positions on the edges of the channels. Accordingly, the Magic router omits the greedy router's channel widening step, reporting failure if a net could not be brought into the channel from some top or bottom pin.

## 5. Routing Switchboxes

The greedy channel router handles pins on at most the top, left, and bottom sides of a channel. To make it a switchbox router, the Magic router contains additional rules to make connections on the right edge of the channel. Furthermore, the Magic router removes the assumption that nets have at most one pin on each end of the channel.

The Magic router deals with switchbox connections by introducing the notion of *reserved* tracks. A track is reserved if it is needed by some net to make a connection on the right edge of the channel. When approaching the end of the channel the router makes vacating jogs to clear reserved tracks and then jogs the appropriate nets into these tracks (Figure 8). Additionally, after nets with only one right edge pin have made their last top and bottom pin connections, their right edge tracks become reserved, other nets vacate these tracks, and the router

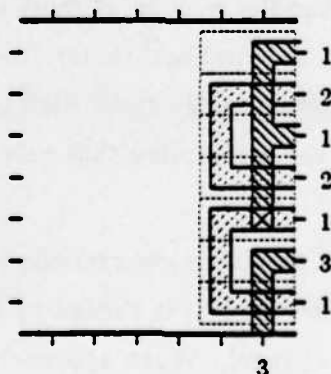




**Figure 8.** The outlined areas are reserved for nets making connections at the end of the channel. Any other nets entering these areas make vacating jogs, allowing the required nets to occupy the tracks.

tries to jog nets into their final tracks. Vacating reserved tracks uses the same mechanism provided to vacate obstructed tracks.

If a net has more than one pin on the right edge of the channel, the router needs to split the net to connect to them. Split nets occupy tracks that could otherwise be used to help route the channel. Therefore splitting to make multiple end connections is only done when the router gets close to the end of the channel. *Close* is a parameter the user sets to control net splitting. A typical value is two columns.



**Figure 9.** As the router approaches the end of the channel, nets with all of their pins on the right edge of the channel require tracks to be assigned to the nets. This is done if at least two tracks can be allocated and joined with vertical wiring.

Nets with all of their pins on the right edge edge of the channel are another

complication. As the router nears the right edge of the channel it has to decide when to first assign tracks to these *right edge* nets. Since there are no connections to previous pins, a right edge net is introduced into the channel only if it can be assigned to at least two tracks that can be joined by vertical wiring (Figure 9).

We carry this one step further. Groups of two or more tracks for a particular right edge net may be introduced into the channel, even if the groups themselves can not immediately be joined. The task of joining these groups is easier, since the top track of one group need only be connected to the bottom track of a net's higher group.

## **6. The Magic Routing Algorithm**

The Magic router operates in three phases. It begins by making a pre-routing scan of the routing area, identifying obstacle thresholds. After identifying obstacle thresholds, the router extends nets from left edge pins into the routing area and routes it using the column-by-column scan. After routing the channel the Magic router invokes a post processing step to maximize metal and reduce vias.

### **6.1. Finding Obstacle Thresholds**

Obstacle thresholds are generated for all multi-layer obstacles and some single layer obstacles. Multi-layer obstacles such as contacts, crossings, and poly/metal edges must always be avoided as tracks extend from left to right, since it is not possible to bridge these obstacles in any layer. Single layer obstacles extending horizontally for more than one column's width also generate threshold areas. Single layer obstacles extending horizontally for only one column's width do not generate thresholds since the vertical wiring gained in the obstructed area is offset by the vertical wiring wasted in jogging around the obstacle.

Depending on the height of the obstacle, many nets may have to be jogged around it. Not all nets can make vacating jogs in the same column because the vertical wiring for one vacating jog blocks another net from making its vacating jog. On the other hand, vacating tracks long before they near obstacles wastes channel routing area. In recognition of this, the Magic router makes vacating jogs

around an obstacle depending on how far away and how high the obstacle is. Higher obstacles, which block more tracks, cause nets to start vacating jogs farther away, while shorter obstacles can be approached more closely before vacating jogs commence. The width of the threshold is the product of a parameter, *obstacle threshold constant*, and the height of the obstacle. This parameter allows some control over how soon the router attempts to vacate obstructed tracks. A typical value for this parameter is 1.

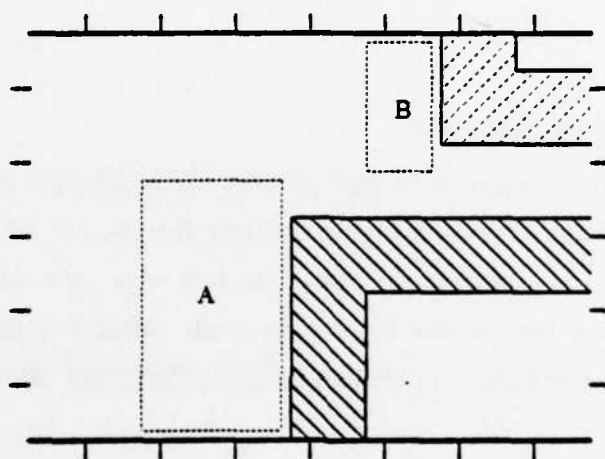


Figure 9. Taller obstacles may require more nets to vacate their thresholds; therefore taller regions have wider thresholds.

The obstacle threshold also extends one track above and below the obstacle. Nets do not get assigned to these tracks unless no other track is free. This allows contacts to be placed if vertical wiring has to switch layers to bridge the obstacle (Figure 7).

## 6.2. Wiring Rules

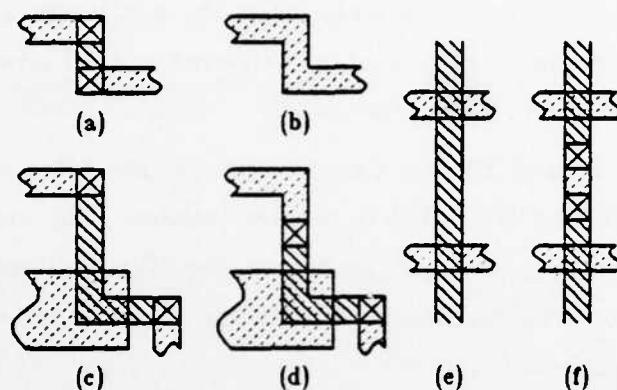
This section presents the set of rules the Magic router uses to control the placement of contacts and vertical jogs. The following discussion omits details that are identical in the greedy router. The rules are:

- a. *Place Track Contacts:* As the first step in wiring a column, place a contact in each unobstructed track, if either the next column or the previous column has an obstruction in the preferred horizontal track layer. The contact serves one of three purposes: (a) it switches the net from the preferred

horizontal track layer (metal) to the alternate layer (poly) when the net enters a river-routed region; (b) it switches the net from the alternate layer back to the preferred horizontal layer when the net leaves a river-routed region; or (c) it switches the track to the preferred vertical layer in preparation for jogging the net to another track.

- b. *Make Minimal Top and Bottom Connections:* Do not bring a net into an unobstructed track that is blocked in the next column. This step may bring a net into an obstructed track. If this occurs, step (f) will attempt to jog the net to an unobstructed tracks. Report failure if some net could not be brought into the channel.
- c. *Collapse Split Nets.*
- d. *Reduce the Range of Tracks Assigned to Split Nets:* Do not move a net from a free track to a track that needs to be vacated.
- e. *Raise Rising Nets and Lower Falling Nets:* Do not jog from a free track to one that needs to be vacated.
- f. *Vacate Obstructed Tracks:* Identify tracks from which nets should be vacated. These are tracks which are either in the threshold of an obstacle or are reserved to make some end connection. Try to vacate to the nearest empty, unobstructed track. Do not vacate to another obstructed or reserved track unless the source track is blocked (ie. runs into a multi-layer obstacle) and the destination track is not blocked. Give preference to vacating jogs that move rising and falling nets closer to their next pin.
- g. *Split Nets to Make Multiple End Connections:* If within *channel end constant* columns of the end of the channel, attempt to split nets to make multiple connections at the end of the channel. This is the opposite of the collapsing step c above. The best pattern is that which splits the most tracks.
- h. *Extend Active Tracks to the Next Column:* Report an error if some track is prevented from extending into the next column by the presence of a multi-layer obstacle that could not be avoided.

### 6.3. Metal Maximization

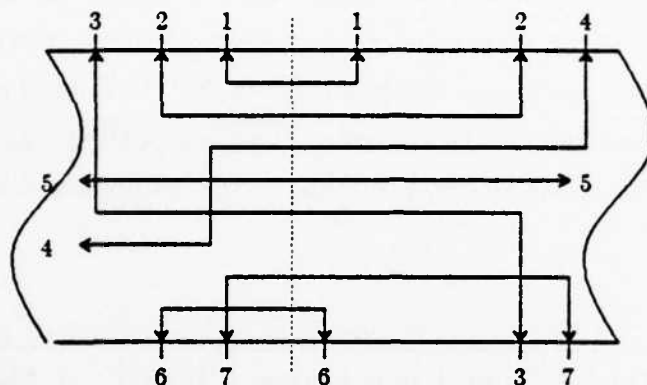


**Figure 11.** A postprocessing step maximizes metal. This may delete or move vias. It may also introduce vias.

After the subchannels are routed, the Magic router concludes with a metal maximization step. (Figure 11). Since the router already routes metal horizontally wherever possible, this step replaces vertical wiring in polysilicon with vertical wiring in metal, subject to constraints imposed by obstacles in the channel. Vias are deleted wherever they become unnecessary.

## 7. Channel Splitting

The Magic router also extends the greedy router by including a channel splitting feature. It splits a channel in two at a point of maximum density, assigns tracks to nets crossing the split, then routes both subchannels outwards from the column of the split. The intent of channel splitting is to improve the routability of the two resulting subproblems by (1) assigning tracks to the nets crossing the split to remove conflicts between vertical wiring, and (2) removing split nets at the column where the channel is divided, to guarantee that there are enough available tracks to accommodate the nets that must cross this column. Channel splitting is done if the length in columns of each of the resulting subproblems is greater than or equal to the parameter *minimum channel size*, and if the density of the routing problem is close to the size of the channel. If the channel can not be split, then the router routes it from left to right or from right to left, at the discretion of the user.



**Figure 12.** To increase the routability of the two subchannels the router assigns tracks to nets crossing the split. Nets are ordered based on their rising/falling status and the distance to their closest left and right pins.

Channel splitting is not recursive -- it is done at most once. The idea is to route away from the point of maximum density. Splitting each subchannel at its point of maximum density would result in subchannels routing from one highly constrained region to another.

After deciding where to split the channel, the Magic router assigns tracks to the nets crossing the split. The ranking procedure assigns each net a ranking number which is the average of the distance from the center track of the channel to the net's target tracks in the left and right subchannels. The top tracks go to nets which rise to pins on the top edge of both subchannels. The bottom tracks are assigned to nets which fall to pins on the bottom edge of both subchannels. All other nets, including those rising or falling an intermediate distance, and those steady in both subchannels, get distributed between the first two groups.

Another discriminator is used among nets rising to the top or falling to the bottom of both subchannels. A net *a* ranks above another net *b* if both *a*'s nearest left pin and its nearest right pin are closer to the split column than *b*'s corresponding pins. If the distances overlap (ie. *a*'s left pin is closer than *b*'s, and *b*'s right pin is closer than *a*'s), then the net with the smaller sum of distances is placed above the other. A similar procedure is used for falling nets. The intent is to order the nets to eliminate crossings wherever possible. If nets must cross, this procedure favors the net traveling the shorter distance.



### 8. Implementation and Performance

For channels without obstacles the Magic router produces results similar to those produced by other good channel routers such as the hierarchical router [BuP], the greedy router [RiF], and Algorithm #2 [YoK]. In spite of omitting the track insertion step from the greedy algorithm, it routes Deutsch's difficult in the same number of tracks as the the greedy router. The results are summarized in Table 1.

Router	Tracks	Vias	Wire Length	Time (sec)	Machine
Magic (no obstacles)	20	376	4099	1.5	DEC VAX 11/780
Magic (with obstacles)	20	376	4099	3.0	DEC VAX 11/780
Algorithm #2	20	-	-	2.1	DEC VAX 11/780
Greedy	20	347	4150	7.93	DEC KA-10
Hierarchical	19	270	3983	24	IBM 370/3033

**Table 1.** Router Results for Deutsch's Difficult Example

Most of the numbers in Table 1 were taken from [BuP]. The first table entry refers to our implementation of a modified greedy switchbox router before obstacle avoidance was added. The reported number of vias for the Magic router does not show the results of metal maximization.

The table shows that the Magic router is competitive with other channel routers on conventional routing problems. It produces nearly optimal solutions quickly, which may be more valuable in practice than programs such as the Hierarchical router which produce slightly better results after significantly greater computation. Adding obstacle avoidance nearly doubled the running time of our router.

Our figures provide a good comparison between Yoshimura and Kuh's Algorithm #2 and Rivest and Fiduccia's greedy router. Rivest and Fiduccia's router



was implemented in LISP on a KA-10. The Magic router without obstacle avoidance (which is almost identical to the greedy router) is implemented in the *C* programming language. Algorithm #2 is implemented in FORTRAN. Both the Magic router (without obstacle avoidance) and Algorithm #2 run on VAX 11/780s running Berkeley Unix. The early version of our router runs faster than the already fast Algorithm #2, and produces a result using the same number of tracks.

Experience with channel splitting has so far been disappointing. It has turned out to be useful mostly for assigning crossings in river routed regions. In other cases splitting the channel typically increases the number of tracks required to route the channel. Better rules for ordering the nets crossing the boundary between the subchannels might change this. Another idea would be to use different criteria to decide where to split the channel.

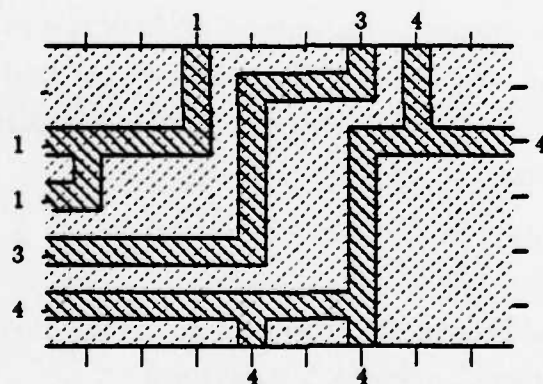


Figure 13. The Magic router river-routes in areas completely blocked in a single layer.

As an example of the range of problems handled by the Magic router, Figure 13 shows a channel completely covered with metal. Our router does a reasonable job of routing this problem.

Postprocessing to increase metal and remove vias appears to significantly improve the quality of the routing.

## 9. Conclusions

Our obstacle avoiding channel router adds flexibility to our design environment. It allows designers to route critical signals by hand or with separate routing steps. After critical signals are routed, the router makes the remaining connections.

The Magic channel router provides this obstacle avoiding capability, while also considering tradeoffs and interactions between nets. It accomplishes this using a rule based, column sweep routing algorithm which is simple, flexible, and fast. The simplicity of this approach makes it an attractive vehicle for further experimentation.

## 10. Acknowledgements

Robert Mayo, Walter Scott, and George Taylor all participated in discussions resulting in this work and provided comments on drafts of this paper. Mark Hill, Randy Katz, Carlo Sequin, and David Wallace also reviewed drafts and provided helpful comments. The work described here was supported in part by SRC under grant number SRC-82-11-008.

## 11. References

- [BuP] Burstein, M., and Pelavin, R., "Hierarchical Channel Router", *Proc. 20th Design Automation Conference*, Miami (1983)
- [Che] Chen, H., Private communication with authors.
- [CHK] Chen, N. P., Hsu, C. P., and Kuh, E. S., "The Berkeley Building-Block Layout System for VLSI Design", ERL memo UCB/ERL M83/10, University of California at Berkeley, (Feb. 1983).
- [Lee] Lee, C. Y., "An Algorithm for Path Connections and its Application", *IRE Transactions on Electronic Computers*, pp. 246-365 (September 1961).
- [Hig] Hightower, D., "A Solution to the Line Routing Problem on the Continuous Plane", *Proceedings Design Automation Workshop*, pp. 1-24, (1969).
- [OHM] Ousterhout, J. K., Hamachi, G. T., Mayo, R. N., Scott, W. S., and Taylor, G. S., "Magic: A VLSI Layout System", In this technical report.

- [Riv] Rivest, R. L., "The 'PI' (Placement and Interconnect) System", *Proc. 19th Design Automation Conference*, Las Vegas (1982).
- [RiF] Rivest, R. L., and Fiduccia, C. M., "A Greedy Channel Router", *Proc. 19th Design Automation Conference*, Las Vegas (1982), pp. 418-424.
- [Sou] Soukup, J., "Circuit Layout", *Proceedings of the IEEE*, Vol. 69, No. 10 (Oct. 1981), 1281-1304.
- [YoK] Yoshimura, T., and Kuh, E. S., "Efficient Algorithms for Channel Routing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-1, No. 1, (Jan 1982).

# Magic's Incremental Design-Rule Checker

George S. Taylor and John K. Ousterhout

Computer Science Division  
Electrical Engineering and Computer Sciences Department  
University of California  
Berkeley, California 94720

## ABSTRACT

The Magic VLSI layout editor contains an incremental design-rule checker. When the circuit is changed, only the modified areas are rechecked. The checker runs continuously in background to keep information about design-rule violations up-to-date. This paper describes the basic rule checker, which operates on edges in the layout, and the techniques used to perform incremental checking on hierarchical designs.

**Keywords and Phrases:** design-rule checking, interactive layout editor

## 1. Introduction

Almost all existing design-rule checking (DRC) programs are batch oriented [1] [2]. They read in a complete circuit layout and check the entire design. If the circuit is changed, the only way to find out whether design rules have been violated is to recheck the entire design, no matter how small the change or how large the design. For chips with tens of thousands of transistors, a batch DRC run may require many hours of computer time.

This paper describes a different approach to design-rule checking. As part of the Magic VLSI layout editor [3], we have built a checker that operates *incrementally*. When the layout is modified, Magic records which areas have changed and rechecks only those areas. While the user continues editing, the checker runs in background and highlights errors as it finds them. There is no set-up time because the checker works from the same data structure used to represent the layout. Since most changes made with the interactive editor are small and the checker is fast, it can usually display errors instantly.

The user's view of design-rule checking is a simple one. As he edits the circuit, small white dots appear over areas that contain layout errors. As soon as the errors are fixed, the white dots go away. Error information is stored with the design and will reappear during the next editing session if the violation has not been fixed. This information is always kept up-to-date, so there is never any need to run a batch checker.

In the next section, we describe Magic's internal representation for a layout and explain how particular features contribute to fast incremental checking. Section 3 describes how the basic checker works from edges in the layout and how design

rules are specified. Section 4 shows how we use the basic checker for incremental checking of individual cells, and Section 5 describes how hierarchical designs are handled. Section 6 gives measurements of the checker's speed.

## 2. Representation of a Layout

In Magic, a layout is represented as a hierarchical collection of cells. Each cell contains mask information plus pointers to subcells. For now, we will consider only a single cell at a time (Section 5 generalizes the solution to handle hierarchical designs).

Magic represents the mask layers of a cell with rectangular *tiles*, which means that it handles only Manhattan geometries. Each tile indicates the type of mask layer it represents. Tiles are connected to form *planes* by a technique called *corner-stitching* [4] illustrated in Figure 1. The tiles in a plane are non-overlapping and cover it completely. Empty areas are covered with tiles of type "space."

Each cell contains several planes of mask information. Mask types that interact (such as polysilicon and diffusion) are stored together in the same plane, while those that do not interact (such as polysilicon and metal) are stored in different planes. Contacts between mask types on different planes are represented in both of them. Our nMOS process has two planes: one for metal and one for polysilicon, diffusion, and transistors.

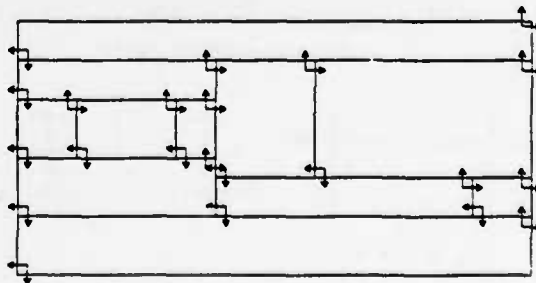


Figure 1. An example of a corner-stitched plane. Each plane contains tiles of different types that cover the entire area of the plane (space tiles are used where there is no mask material). Each tile contains four pointers that link it to neighboring tiles at its corners. The pointers make it easy to find all the material in a given area.

Instead of working directly with physical mask layers, Magic uses *abstract layers* to represent structures such as transistors and contacts. The abstract layers appear in the database as tiles with special types. For example, instead of representing an enhancement transistor as a polysilicon tile over a diffusion tile, it is represented with a tile of type "enhancement transistor." A more complete explanation of the abstract layers is given in [3]. What matters here is that all the interesting features are represented explicitly: there is no need to cross-register diffusion and polysilicon to discover the transistors.

The design-rule checker takes advantage of Magic's database in three ways. First, the corner-stitched tiles allow DRC to find material in a given area very quickly. Second, division of mask information into planes allows the checker to work with one plane at a time, ignoring irrelevant geometry on other planes. Third, there is no need to extract features by registering layers: the abstract layers represent the important features explicitly. Because of these features, there is no need for the checker to manage a separate structure of its own: it works directly from the layout database.

### 3. The Basic Checker

This section describes the basic design-rule checking paradigm used to validate an area of a single corner-stitched plane. Later sections show how this basic checker is used to perform incremental checks on a single cell, and then on a hierarchy of cells.

#### 3.1. Edge-based Rules

Magic's design rules are based on edges between tiles. Each rule can be applied in any of four directions, as shown in Figure 2. The rule table contains a separate list of rules for each possible combination of materials on the two sides of an edge. In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on *type2*'s side of the edge. This area is referred to as the *constraint region*.

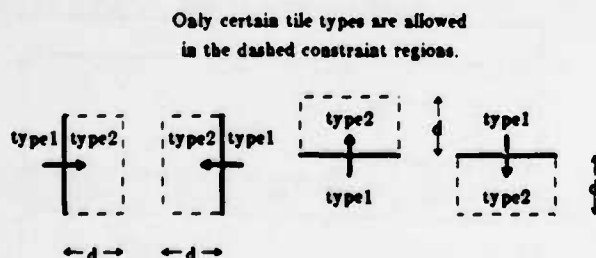


Figure 2. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of *type1* and *type2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance *d* on *type2*'s side of the edge.

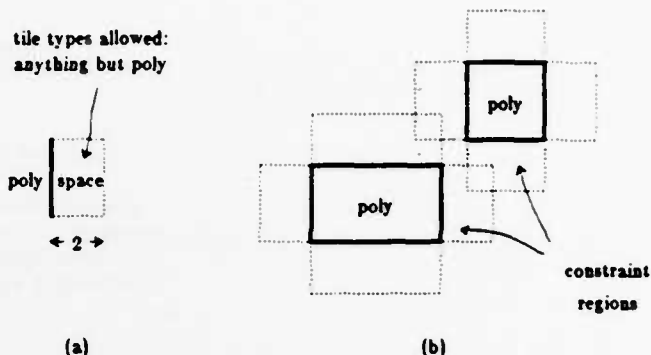


Figure 3. If only the simple rules from Figure 2 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

Unfortunately, this simple scheme will miss errors in corner regions, such as the case shown in Figure 3. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 4 for an illustration of the corner rules and how they work. Table 1 gives a complete summary of the information in each design rule.

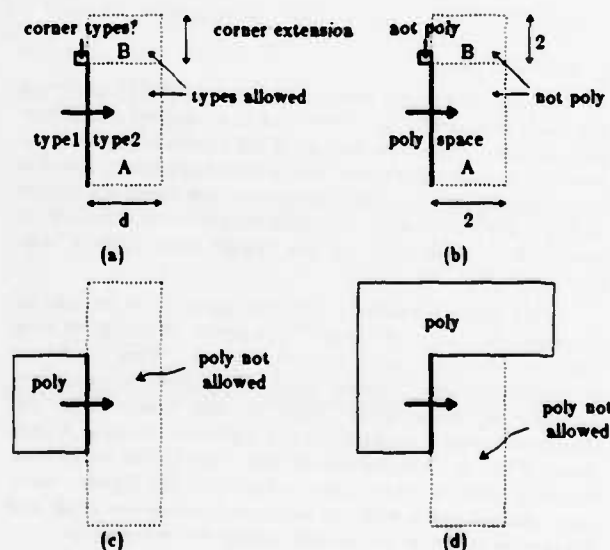


Figure 4. The complete design rule format is illustrated in (a). Whenever an edge has *type1* on its left side and *type2* on its right side, the area *A* is checked to be sure that only *types allowed* are present. If the material just above and to the left of the edge is one of *corner types*, then area *B* is also checked to be sure that it contains only *types allowed*. A similar corner check is made at the bottom of the edge. Figure (b) shows a polysilicon spacing rule, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge.

Parameter	Meaning
<i>type1</i>	Material on first side of edge.
<i>type2</i>	Material on second side of edge.
<i>d</i>	Distance to check on second side of edge.
<i>layers allowed</i>	List of layers that are permitted within <i>d</i> units on second side of edge.
<i>corner types</i>	List of layers that cause corner extension.
<i>corner extension</i>	Amount to extend constraint area when corner type matches.

Table 1. The parts of an edge-based rule.

### 3.2. Applying the Rules

To check a portion of a single plane, Magic must first find all of the edges in that area. This is accomplished by searching for all of the tiles in the area. The corner-stitched data structure is well suited to searches of this sort: see [4]. For each tile, the checker examines its left and bottom sides (the top and right sides of the tile will be checked by the neighbors on those sides). Since the tile may have neighbors of different types on the same side, the checker searches through all the neighbors to divide the side of the tile into edges with a single material on each side.

To process an edge, the mask types on each side of it are used to index into the rule table to find the list of rules for that kind of edge. Each rule in the list is checked, and error information is recorded for any areas where the constraints are not satisfied. For each edge there are two rule applications: left-to-right and right-to-left (for vertical edges) or bottom-to-top and top-to-bottom (for horizontal edges). A different list of rules is applied in each direction, since the mask types are reversed.

### 3.3. Specifying Design Rules

Design rules are specified in a technology file that contains the rules and other technology-specific information. When Magic starts executing, it reads this file and builds the rule table. Initially we specified rules in the detailed form of Table 1, with one line for each edge rule. This scheme proved to be unworkable, because there were many rules and it became difficult to convince ourselves that the rule set was complete and correct.

In order to simplify the process of creating rule sets, Magic now permits rules to be specified with high level macros for width and spacing. For example, the macro

```
spacing efet,dfet dmc,pmc 1
```

is expanded into several rules to verify that enhancement and depletion transistors are always separated from diffusion-metal contacts and poly-metal contacts by at least one unit. The macro

```
width poly,pmc,buried,efet,dfet 2
```

is expanded into the set of edge rules needed to verify that any region containing any of the five mask types listed is always at least two units wide.

Most of the rules for our processes are simple width and spacing checks, so these two macros considerably simplify the writing of rule sets. Our nMOS rule set contains 8 width rules, 6 spacing rules, and 10 of the detailed edge rules for situations that cannot be handled by the width and spacing rules (e.g. transistor overhangs). Magic expands these 24 high-level rules into 127 detailed edge rules. Our CMOS process requires 35 high-level rules that are expanded into 188 detailed edge rules.

The width and spacing macros make Magic's checker more efficient because the width and spacing rules are symmetric. If layers *x* and *y* are too close together, the violation can be detected from either an edge of *x* or an edge of *y*. This means that it is unnecessary to check the rules from both edges. Magic takes advantage of this symmetry by checking width and spacing rules in only two directions (left-to-right and bottom-to-top). In addition, symmetric rules mean that corner extension is only necessary on one end of each edge. Since most of the detailed edge rules come from the width and spacing macros, this speeds up the checking process by almost a factor of two.

Magic's design-rule language has certain limitations. It can only express constraints that depend on a limited amount of local context. One example of a rule that depends on more extensive context is a rule where the spacing between adjacent parallel wires depends on their length. Another example is a reflection rule where the minimum size of one material depends on its proximity to another material. In the processes that we use, complex rules such as these are replaced with more conservative, but simpler, rules.

## 4. Continuous Design-Rule Checking

This section shows how the basic checker is used to provide continuous incremental rule validation. As in the previous section, we consider only single-cell designs here.

In order to perform DRC incrementally, Magic maintains two extra kinds of information with each cell, stored in the same form as mask layers. First, Magic keeps information about rule violations that have been detected but haven't been corrected. The violations are represented by error tiles that cover the areas where rule constraints are not satisfied. The second kind of information consists of tiles describing the areas of the circuit that need to be reverified. The error tiles and the reverify tiles are stored in separate corner-stitched planes. Each cell contains its own error and reverify planes.

When a designer changes a cell, Magic creates reverify tiles that cover the area modified. The design-rule checker runs in background while Magic is waiting for the designer to enter the next command. DRC first searches for reverify tiles. Then it invokes the basic checker over the area covered by each tile found. The basic checker reverifies the area on each of the cell's planes, updates error tiles, and erases the reverify tile. Changes to the error information are reflected on the graphics screen.

If the designer invokes a command while the checker is running, the checker stops so that the command can be processed without delay. After the command finishes, the checker resumes by starting over on the area that it was working on just before the interruption. Large reverify tiles are broken up into small ones before checking, in order to reduce the amount

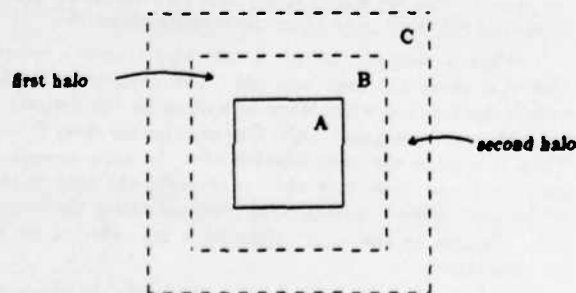


of work that might have to be repeated. When there are large areas to be reverified, the checker works across the design in a style like "Pac-Man," gobbling up reverify tiles and spitting out error tiles.

If incremental checking is done carelessly, errors may not be detected when new violations are introduced, or error information may be left in the database even after the violations have been corrected. Figure 5 illustrates the problem and Magic's solution. When an area is modified, error information may be affected in both the area that was modified and in the surrounding area (for example, material in area A may be too close to something in the surrounding area B). We call the surrounding area the *halo*. Its width is equal to the largest distance in any design rule. Error information must be recomputed in the modified area and its halo. However, errors in the halo don't necessarily involve the inner modified area. They may come from interactions between the halo and a second halo outside it. To regenerate errors in the first halo correctly, information in the second halo must be considered.

When area A of Figure 5 is modified, Magic rechecks it by deleting all error information in A and B. The checker then generates new error information in both areas by invoking the basic checker over areas A, B and C. Any errors found during this process are clipped to the area of A and B, so that error information is not affected outside the region where errors were erased.

The reverify and error tiles are stored with cells so that they are not lost at the end of an editing session. Normally, there will be no reverify tiles left at the end of a session, but if a large area has been changed recently, it is possible that it won't have been reverified when the session ends. In this case, the reverify tiles are written to disk with the cell. When the cell is read in during the next editing session, the design-rule checker will notice the reverify tiles and continue the reverification process. The reverify and error tiles are identical to the tiles used to represent mask layers, except that they are not manipulated directly by the designer.



**Figure 5.** If area A is modified, the design-rule checker erases existing error information in both A and B. Errors in B could have come from information in A, B or C, so all three areas must be checked to regenerate all of the errors. The width of the halos B and C is equal to the largest distance in any design rule.

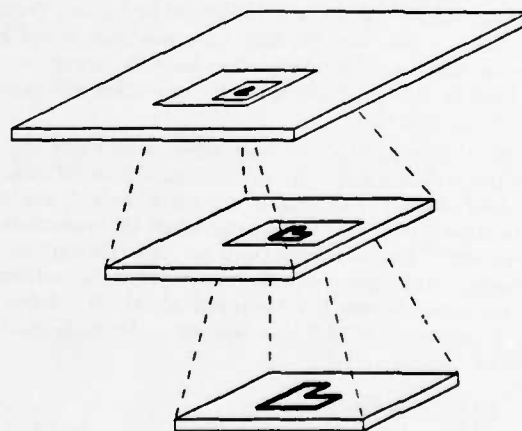
## 5. Hierarchical Checking

Most of the layouts created with Magic consist of hierarchical cell structures rather than single cells (Figure 6). Each cell may contain subcells, and the subcells may overlap other subcells or mask information in the parent. A subcell may appear any number of times in any number of parents.

In hierarchical designs, errors can arise in any of three ways:

- the mask information of an individual cell may be incorrect;
- a subcell may interact incorrectly with another subcell; and
- a subcell may interact incorrectly with mask information in its parents.

Magic's incremental checker includes facilities to detect all of these errors. Overlapping subcells are no more difficult to handle than subcells that merely abut, since interaction errors are possible in either case.



**Figure 6.** Circuits are defined by cells arranged in a hierarchy. If mask information is changed in a low-level cell, Magic checks to be sure that the cell is consistent by itself and that there are no illegal interactions in parents or other ancestors.



### 5.1. Simple Checks and Interaction Checks

Two overall rules guide the hierarchical checker. First, the mask information in every cell must satisfy the design rules by itself, without consideration of subcells. Second, each cell and its subcells must together satisfy all the design rules, without consideration of how that cell is used in its parents. If the layout is viewed as a tree structure, the first rule means that each node of the tree must be consistent, and the second rule means that each subtree must be consistent.

The overall rules result in two kinds of design-rule checking. The first rule is verified by running the basic checker over the planes containing mask information for each cell; this is called a *simple check*. The second rule is verified with an *interaction check* that considers interactions involving subcells. Each cell uses separate planes to hold its mask information, so interaction checks must combine information from different planes.

To make an interaction check on an area, the hierarchical structure is "flattened" to produce a new set of corner-stitched planes that combines all the information from all cells in the area to be checked. This includes mask information from the parent cell, plus mask information from subcells and sub-subcells, and so on. Once all the mask information in the area has been collected into a single set of planes, the basic checker is invoked on these planes in the standard fashion (halo expansion is performed as described in Section 4). Errors arising from the interaction check are placed in the parent cell.

Interaction checks are more expensive than basic checks, since they involve flattening a piece of the hierarchy. Fortunately, interaction checks can often be avoided. For example, if an area contains no subcells, then there is no need to perform an interaction check on that area. A simple check will find all errors. The interaction check can also be avoided if there is only a single subcell in an area, with no other subcells or mask information nearby. In this case any errors must come from within the subcell, and those errors will be found by checks made within that cell. Interaction checks are necessary only in areas where a subcell is within one halo distance of mask information or another subcell. Even then, we only need to check the area around the interaction.

### 5.2. Checking Upward In the Hierarchy

When a cell is modified, simple checks and interaction checks have to be performed within that cell, and also within its parents in the hierarchy. For example, suppose mask information has been edited within a cell. Then a simple check must be performed within that cell, as well as an interaction check if there are subcells near the modified area. However, these two checks are not sufficient. If the modified cell is a subcell of other higher-level cells, then the change may have introduced interaction problems within the higher-level cells. For each parent of the modified cell, an interaction check must be performed over the area of the modification. Interaction checks must also be performed in grandparents, and so on up to the top-level cell in the hierarchy. In the cell that was modified, both simple and interaction checks must be performed, but in the parents and grandparents only interaction checks are necessary.

Magic uses two kinds of verify tiles to handle the two kinds of checks. When a cell is modified, "verify-all" tiles are placed in that cell to signify that both simple and interaction checks must be performed. At the same time, "verify-interactions" tiles are placed in parents and grandparents to indicate that interaction checks have to be performed. The background checker keeps track of which cells in the database contain verify tiles and performs each kind of check wherever necessary.

In the worst case, the hierarchical algorithm could result in the modified area being rechecked once at each level of the hierarchy above the cell that was changed, with a separate flatten operation required for each check. However, in deep hierarchies most of the interaction checks are avoidable: in cells far above the modified one, the modified area will almost certainly appear in the middle of a single subcell with no mask information or other subcells nearby. Unless there are many large subcell overlaps, any given area of mask information is likely to require an interaction check at only one point in the hierarchy.

### 5.3. Arrays

One other form of hierarchical check arises because Magic has an array construct. To simplify the creation of cell arrays, Magic contains a special array facility: each subcell may consist of either a single instance or a one- or two-dimensional array of identical instances. Because of the array construct, there is actually a third overall rule that guides the hierarchical checker: each array must satisfy all the design rules, independently of other information in the parent containing the array. Whenever a change is made to an array, the array structure is reverified by checking the three areas shown in Figure 7.

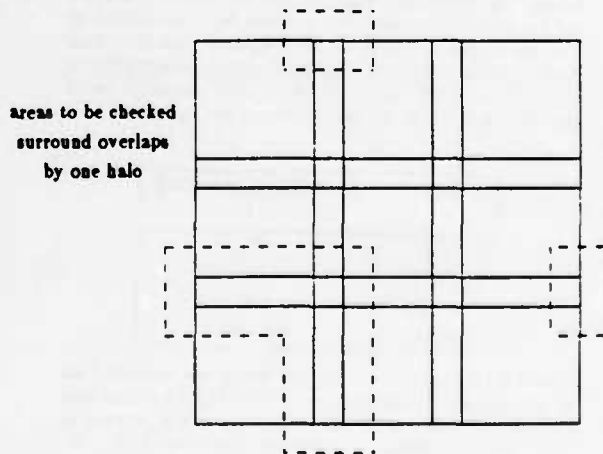


Figure 7. The elements of this 3 by 3 array overlap in both the horizontal and vertical directions. The array is internally consistent if the three dotted areas satisfy the design rules. All possible interactions between elements of the array are identical to the ones that occur in these three regions.

## 6. Implementation and Performance

The design-rule checker is written in C. Its 2800 lines of code are divided into roughly equal fifths for the basic flat checker, continuous checking, hierarchical checking, building the internal rule table from the technology file, and the command interpreter.

The basic checker processes 2000 tiles per second on a VAX 11/780 running Unix. Measurements of the number of edges found and rules checked per tile are given in Table 2 for a typical flat cell and a worst-case cell hierarchy. Table 3 compares Magic's performance with that of other systems, based on transistors per second. The number for Magic was derived from actual designs that used between 20 and 30 tiles per transistor.

A typical change to a circuit involves only a few tiles, so the cost of incremental reverification is dominated by the size of the halos. From this, we estimate that roughly 50 tiles have to be checked per command in an nMOS design. This requires about one-fortieth of a second of CPU time.

Cell	Tiles	Tiles / second	Edges / tile	Rules / edge
RISC floor	64585	2000	0.9	1.4
ALU latch	7485	590	1.9	1.6

Table 2. Performance measurements for Magic's design-rule checker. The column for edges/tile shows how many edges between two different materials were found per tile. (This number can be less than one because adjacent tiles can have the same type.) The last column shows the average number of rules applied across each of these non-trivial edges. The RISC floor plan contains the top level of routing for a microprocessor, but no subcells. The number of tiles per second is typical for flat checking. The ALU latch example consists on a cell copied on top of itself. This gives a worst-case speed for hierarchical checking, where the limiting factor is the time to flatten the hierarchy.

System	Transistors / second
Lyra [2]	2
Baker [1]	3
Mart [5]	6-8
Magic	60-100

Table 3. Performance of several design rule checkers. All of the programs were run on a VAX 11/780. Lyra uses corner-based rules, sorts tiles into bins, and is written in Lisp. Mart is similar to Lyra, but is written in C. The Baker checker uses a raster-scan approach and is also written in C. Unfortunately, we were not able to obtain corresponding results for industrial design rule checkers.

## 7. Conclusions

Magic's design-rule checker demonstrates that incremental checking is feasible. We think that circuit designers will find that continuous feedback reduces the time needed to create new designs or modify existing ones. The key to the incremental checker is low overhead: the ability to run from the same database as the interactive editor, the ability to find important edges in the layout quickly, and the ability to find nearby material quickly. The two features of Magic's database that reduce overhead are the corner-stitched tile planes and the abstract mask layers. Extending the checker to work in hierarchical designs frees the designer from tedious reverification of interactions when subcells are revised.

## 8. Acknowledgements

Gordon Hamachi, Bob Mayo, and Walter Scott participated in discussions that led to the incremental checker and provided many useful comments on drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), under Contract No. N00034-K-0251.

## 9. References

- [1] C. M. Baker and C. Terman, "Tools for verifying integrated circuit designs," *Lambda* (now *VLSI Design*) Vol. 1, No. 3 (1980), pp. 22-30.
- [2] M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 18th Design Automation Conference*, June, 1982, pp. 530-36.
- [3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "Magic: A VLSI Layout System," *Proc. 21st Design Automation Conference*, June, 1984.
- [4] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," *IEEE Transactions on CAD/ICAS*, January, 1984, pp. 87-100.
- [5] B. J. Nelson and M. A. Shand, "An Integrated, Technology Independent, High Performance Artwork Analyzer for VLSI Circuit Design," Technical Report VLSI-TR-83-4-1, VLSI Program, Division of Computing Research, CSIRO, Eastwood, SA 5063, Australia, April, 1983.

# Plowing: Interactive Stretching and Compaction in Magic

Walter S. Scott and John K. Ousterhout

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California  
Berkeley, CA 94720

## Abstract

The Magic layout editor provides a new operation called *plowing*, for stretching and compacting Manhattan VLSI layouts. Plowing works directly on the mask-level representation of a layout, allowing portions of it to be rearranged while preserving connectivity and layout-rule correctness. The layout and connectivity rules are read from a file, so plowing is technology independent. Plowing is fast enough to be used interactively. This paper presents the plowing operation and the algorithm used to implement it.

## 1. Introduction

Plowing is a new operation provided by the Magic layout editor [OHMST 84] for stretching and compacting Manhattan VLSI layouts. It allows designers to make topological changes to a layout while maintaining connectivity and layout rule correctness. Plowing can be used to rearrange the geometry of a subcell, compact a sparse layout, or open up new space in a dense layout. In a hierarchical environment plowing also allows cell placement to be modified incrementally without the need for rerouting. To avoid dependence on a particular technology, plowing is parameterized by a set of layout and connectivity rules contained in a technology file.

Conceptually the plowing operation is very simple. The user places either a vertical or a horizontal line segment (the *plow*) over some part of a mask-level representation of the layout, and then gives the direction and the distance the plow is to move. Plowing can be done up, down, to the left, or to the right. (The rest of this paper will assume plowing to the right.) The plow is then moved through the layout by the distance specified. It catches vertical edges (boundaries between materials) as it moves and carries them along with it. Since only edges are moved, material behind the plow is stretched and material in front of the plow is compressed. Figure 1 shows how plowing can be used to open up new space. Figure 2 shows how it can be used for stretching. Plowing can be used to compact an entire cell by placing a plow to the left and plowing right, then placing a plow at the top and plowing down.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

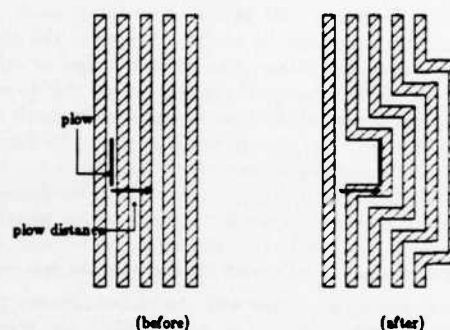


Figure 1. Plowing opens up new space in a dense layout. Geometry is pushed in front of the plow, subject to layout-rule constraints. The connectivity of the original layout is maintained. Jogs are inserted automatically where necessary.

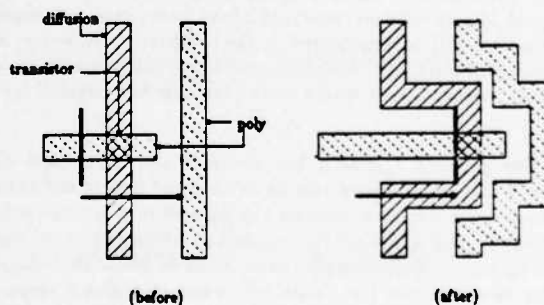


Figure 2. Material to the left of the plow is stretched. Material to the right is compressed. Objects such as transistors do not change in size.

Plowing is so named because each of the edges caught by the plow can cause edges in front of it to move in order to maintain connectivity and layout-rule correctness. These edges can cause still others to be moved out of the way, recursively, until no further edges need be moved. A mound of edges thus builds up in front of the plow in much the same manner as snow builds up on the blade of a snowplow.

Section 2 of this paper discusses plowing in the context of previous work. Sections 3 and 4 introduce the plowing algorithm for a single mask layer. Section 5 extends it to multiple

mask layers and hierarchical designs. Finally, Section 6 presents performance measurements and our experience with plowing in the Magic system.

## 2. Background

VLSI layouts are difficult to modify. Because of this, designers are often committed to the initial choice of implementation, rather than being able to experiment with alternatives. Existing cells often cannot be re-used in subsequent designs because they don't quite fit; it is typically easier to redesign a new cell from scratch than to modify an old one. Bngs in a dense layout are hard to fix, leading to a debugging cycle that can take days or weeks.

Many of these difficulties stem from the fact that seemingly small changes to a layout can have disproportionately large effects. Sometimes this is for electrical reasons. For example, in ratio logic such as nMOS, changes in the size of one transistor may necessitate changes in the sizes of others. However, even purely topological changes—those that preserve the electrical properties of the layout—can require much more work than the size of the change would suggest. As Figure 1 illustrated, merely opening up new space in a layout can cause effects that ripple outward over a much larger area. Rearranging the internal geometry of a cell or modifying the placement of cells in a floor plan can be similarly expensive because of the need to maintain connectivity with the surrounding material.

Previous attempts to cope with the re-arrangement problem have used symbolic design or sticks [RBDD 83, West 81, Will 78]. In the symbolic/sticks approach, designers enter layouts in an abstract form containing wires, contacts, and transistors. The symbolic form is then run through a compactor to generate actual mask information. As part of the compaction, the circuit elements are moved as close together as the layout rules permit. In a symbolic design style, cells can be designed loosely without worrying about exact spacings, since the spacings will be determined by the compactor. However, it is not necessarily any easier to rearrange the topology of a symbolic layout than it would be to rearrange the physical layout.

The plowing approach has many of the advantages of symbolic layout. It allows cells to be designed loosely and then compacted. In addition, plowing can be used to rearrange cells or open up new space, either across the whole cell or in one small portion. Small changes can be made in one area without having to recompact the entire cell, whereas a global recompact may potentially shift all geometry in the cell. The plowing approach lets the designer see the final sizes and locations of all objects as he is editing; in the symbolic approach, it is harder to predict the final structure of a cell from its abstract form, so compaction must be used frequently to see the results of a change to the symbolic form.

## 3. Simple plowing algorithm

Plowing works by finding edges and moving them. An edge is a boundary or a piece of a boundary, parallel to the plow, between material of two different types. When an edge moves, the material to its left is stretched, and the material to

its right is compressed. In this section we will describe how plowing works when only a single mask layer is present. This material will be assumed to have two layout rules: a minimum width of  $w$ , and a minimum separation of  $s$ . Edges will always be boundaries between this material and "empty" space.

The fundamental step in plowing is to move a single edge. The plowing algorithm applies a series of rules to determine which other edges must move as a consequence of this motion. The following discussion presents plowing as though it moves a given edge by first recursively sweeping all other edges out of its way, and then sliding the edge into the newly opened space. Section 4 will present a better scheme for ordering edge motions than this depth-first recursion.

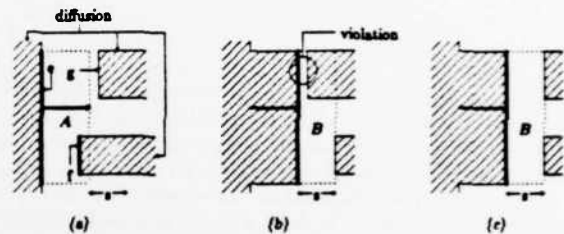


Figure 3. When the edge  $e$  moves, all edges in area  $A$  (the area swept out by  $e$ ) must be moved (a). Moving only these edges results in edge  $f$  moving but not edge  $g$ . This leaves a layout-rule violation (b) between  $e$  and  $g$ . Searching area  $B$  as well as area  $A$  avoids this problem. The two areas are referred to collectively as the *umbra* of edge  $e$ .

### 3.1. Finding edges

Figure 3 depicts a trivial layout consisting of three unconnected pieces of diffusion. The edge labelled  $e$  is to be moved to a final position indicated by the arrowhead. This could be either because  $e$  was caught by the plow, or because it is being moved to make room for some edge to its left. At a very minimum, the rectangular area labelled  $A$  must be swept clear of any material before the edge can be moved. However, because of the spacing rule, any material inside area  $B$  would then be too close to the newly moved edge. Consequently, the area to be swept includes both areas  $A$  and  $B$ . The union of

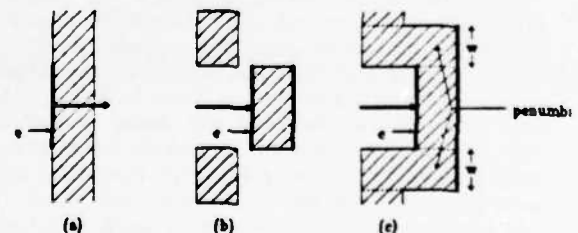


Figure 4. When the edge  $e$  moves (a), edges in its umbra must be moved to the right. If only edges in the umbra are moved, however, the result can be electrical disconnection (b). To avoid this, plowing also moves edges in the *penumbra* to the right, giving the correct result shown in (c). This has the effect of inserting jogs automatically. The height of the penumbra is  $w$ , the minimum width for diffusion. If diffusion had been to the left of  $e$  instead of to the right, the height of the penumbra would have been  $s$ , minimum separation.



these two areas is referred to as the *umbra* of the edge  $e^*$ .

Plowing must also search above and below the umbra to prevent the edge from sliding too close to other edges above or below it. Figure 4a shows why this is necessary. If material were moved out of the umbra alone, as in Figure 4b, the result is electrical disconnection. To avoid this, plowing must also move edges out of the areas above and below the umbra. The correct result is shown in Figure 4c. The areas above and below the umbra are referred to collectively as the *penumbra*. Jog insertion is an automatic consequence of searching the penumbra. Moving edges out of the penumbra also prevents electrical shorts, as can be seen by reversing the roles of material and space in Figures 4a-4c.

The left-hand boundary of the penumbra is not always aligned with the edge being moved. Instead, this boundary is formed by following the outline of the material forming the edge, as illustrated in Figure 5. This insures that the penumbra contains only those edges that must move in order to preserve layout-rule correctness and connectivity. The umbra and penumbra of an edge are collectively referred to as its *shadow*. The shadow of  $e$  contains all the edges that must move as a direct consequence of moving  $e$ .

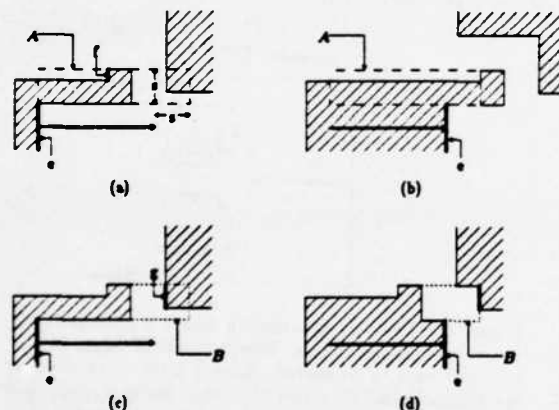


Figure 5. If  $e$ 's penumbra included all of area  $A$ , as shown in (a), then edge  $f$  would be found and moved, resulting in (b). This is undesirable, since  $f$  need not move in order to preserve layout-rule correctness and connectivity. A better definition of the penumbra is area  $B$  only, as shown in (c). Searching this area results in only the edge  $g$  being found and moved, as is necessary to preserve layout rule correctness.

### 3.2. Sliver prevention

The rules described in Section 3.1 guarantee that plowing never moves one vertical edge too close to another. However, they do allow violations to be introduced between horizontal segments that are formed when material is stretched. These violations take the form of slivers of material or space whose height is less than the minimum allowed. Eliminating such slivers requires that their left-hand edges be moved, as illus-

\* In a solar eclipse, the *umbra* is that portion of the moon's shadow from which the sun appears to be completely eclipsed. The *penumbra* is the partial shadow surrounding the umbra. In plowing, the umbra of an edge contains edges directly in its path, while the penumbra contains edges to either side of its path but nonetheless too close.

trated in Figure 6. The left-hand edge of each sliver lies along the left-hand boundary of the penumbra, so it can be found when tracing the outline of the penumbra.

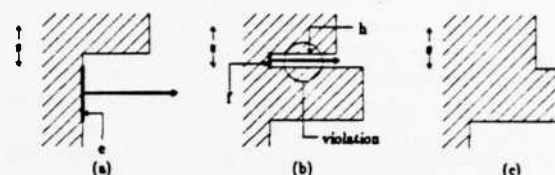


Figure 6. When the edge  $e$  moves (a), a sliver of space is introduced below the horizontal segment  $h$ , as shown in (b). To correct this, the left-hand edge of this sliver,  $f$ , is moved along with  $e$ , but only as far as the right-hand end of the segment  $h$  (c).

### 4. Breadth-first vs. Depth-first Search

In the previous section, plowing was described as a depth-first search in which all edges to the right of a given edge were moved before the edge itself. While this approach is conceptually clear, it has poor worst-case behavior. An  $N$ -tier lattice structure as illustrated in Figure 7 requires on the order of  $2^N$  edge motions, because plowing performs the recursive search to the right of an edge each time the edge is moved. If, as in the example, each edge must be moved once for each of its two neighbors to the left, the edges at the right-hand side of the lattice are moved a number of times that is exponential in the number of tiers.

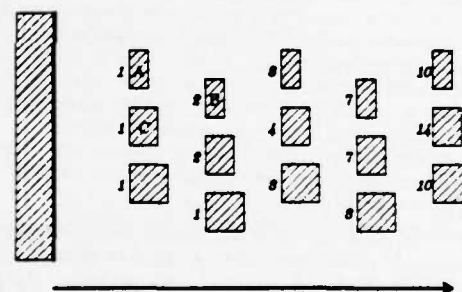


Figure 7. This lattice structure causes exponential worst-case behavior in the depth-first plowing algorithm when edges in the shadow are processed from top to bottom. The objects ( $A$ ,  $B$ , etc.) must be incompressible to cause this worst-case behavior. Object  $B$  is moved once when object  $A$  moves, then slightly farther when object  $C$  moves. The numbers to the left of each object show how many times each of its edges is moved.

Lattice structures such as this one are fairly common in real layouts; a routing channel containing jogs is one example. The real plowing algorithm must avoid paying the exponential cost of plowing such a structure. It does so by waiting until the final position of an edge is known before it performs the search to the right of that edge. This strategy causes the number of edge motions to be linear in the number of edges in the lattice. (See [Oust 84] for a detailed explanation).

A simple way to insure that edges are moved only once their final positions are known is to use breadth-first search. Magic maintains a list of edges to be moved, sorted in order of

increasing  $x$ -coordinate. On each iteration, the leftmost edge is removed from the list and the shadow to its right is searched. Any edges discovered by this search are placed in the list along with the amount they must move. Since the final position of an edge can only be affected by edges to its left, the final position of the leftmost edge in the list is always known.

The original depth-first algorithm allowed the layout to be modified incrementally as plowing progressed, since an edge was never moved until the area into which it was moving had been cleared. Incremental modification is impossible with breadth-first search, since edges to the right will not be moved as long as there are queued edges to the left of them waiting to be moved. Instead of actually updating the layout as it progresses, the breadth-first version of plowing stores with each vertical edge segment the distance it moves. When the shadows of all edges have been searched, and the distance each edge moves has been determined, plowing invokes a post-pass to update the layout from the information stored with each edge.

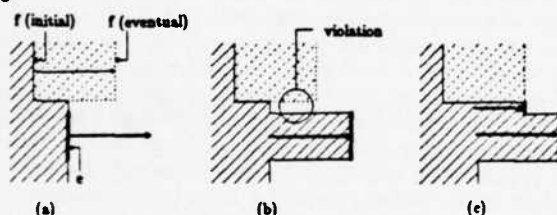


Figure 8. When processing an edge in the breadth-first approach, it is important to use information about the final positions of edges that have already been processed. In (a), it has already been decided to move edge  $f$ , but the edge will not actually be moved until all other edges have been processed. If edge  $e$  is processed without considering the new position of  $f$ , a sliver will result as shown in (b). Instead, the plowing algorithm must consider the eventual positions of edges that have already been processed, to produce the result of (c).

However, if the layout is not modified until all edges have been processed, special care must be taken to avoid the generation of slivers. Figure 8 illustrates the problem. To process each edge correctly, it is important to know what other edges have been already been processed and what their final positions will be. In general, the plowing algorithm must consider edges whose final positions will be in the shadow, rather than those whose initial positions are in the shadow.

The success of the breadth-first algorithm depends on the fact that left-to-right plowing never changes the order of edges along any horizontal line, and never changes any vertical coordinates. Furthermore, each edge has stored with it the distance it is going to move. As a consequence, plowing can use the initial layout structure for searching, and yet can easily find all objects whose final coordinates fall in a given area.

## 5. Extensions for real layouts

This section extends the simple plowing algorithm of the previous two sections to handle multiple mask layers and fixed-size objects such as transistors. It presents a way in which the number of jogs introduced by plowing can be controlled. Finally it describes how hierarchical layouts can be plowed.

### 5.1. Multiple mask layers

The simple version of plowing assumed that the shadow extended to the right of the final position of a moving edge by either  $w$  (the minimum width rule) if material lay to the right of the edge, or  $s$  (the minimum separation rule) if material lay to the left of the edge. This insured that the shadow included all edges directly in the path of the edge being moved. Since the same layout rule applied between the edge being moved and any other edge, all edges found during the search of the shadow would have to move.

With more than one mask layer there may be more than one layout rule to apply for a given edge. For example, in our nMOS process, the minimum separation between diffusion and polysilicon is 2 microns, while that between two pieces of diffusion is 6 microns. Both of these rules apply at an edge between diffusion and empty space.

In section 3.1, the umhbra consisted of the area swept out by an edge being moved, plus an additional area to the right of the final position of the edge. Because several rules may now apply when moving a given edge, the width of this additional area must be the longest distance of any layout rule associated with the edge being moved.

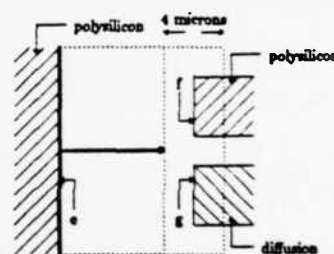


Figure 9. The area of a shadow search is determined by the worst-case layout rule. However, not all edges in that area will have to be moved. Edge  $f$  must move, because the separation between two polysilicon features must be 4 microns and edge  $e$  approaches to within 2 microns of  $f$ . Edge  $g$  need not move since the minimum separation between polysilicon and diffusion is only 2 microns.

However, as Figure 9 illustrates, not all of the edges found while searching the umhbra must actually move. Each edge found must be checked for its minimum allowable separation from the edge being moved. The same techniques used in Magic's layout rule checker [TaOu 84] may be used to perform this check very quickly.

Multiple mask layers require that plowing take extra care to maintain connectivity with material above and below an edge being moved. In the single-layer scheme, the penumbra search guarantees that the material does not become disconnected. However, the penumbra search follows the outline of a single type of material, so it will not by itself guarantee that two adjacent materials of different types will remain connected (see Figure 10).

Special actions must be taken during the penumbra search to handle horizontal edges between different materials. First, if two materials share a horizontal edge, then Magic guarantees that one material does not slide past the end of the other: it maintains a minimum-width connection between the

two (this is the case between materials A and B in Figure 10). Second, if one material completely covers the edge with another material (for example, the A-C edge in Figure 10), Magic plows the other material as much as is needed to maintain complete coverage. This ensures, for example, that transistors are not uncovered by plowing polysilicon off one side.

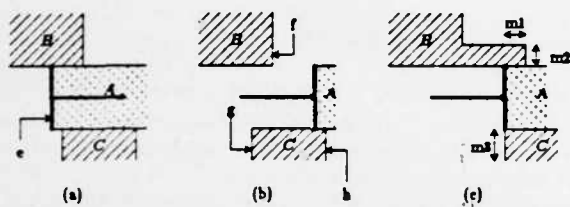


Figure 10. If edge  $e$  is plowed, material A may disconnect from B and C. To prevent this, a minimum-width segment of edges  $f$  and  $g$  is dragged along with  $e$ . The edge  $g$  is moved not to maintain connectivity (which would have been achieved by moving  $A$ ), but to prevent C from being uncovered. In (c),  $m1$  is the lesser of the minimum widths for A and B,  $m2$  is the minimum width for B, and  $m3$  is the minimum width for C.

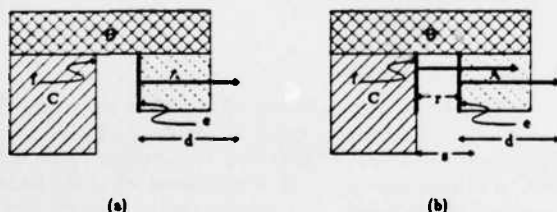


Figure 11. When inelastic objects are present, plowing may have to cope with circular dependencies. Material B is inelastic, and A and C are both minimum-width. When edge  $e$  moves by distance  $d$  in (a), object B must move by the same distance to prevent A from being uncovered. To prevent C from being uncovered, C's left-hand edge must move, finally causing edge  $f$  to move by distance  $d$ . Edge  $e$  is in  $f$ 's shadow as a result, but should not be moved a second time.

## 5.2. Inelastic features

Certain features in a layout should not be stretched or compacted. Transistors, for example, have sizes chosen for electrical reasons, as do contacts. Our discussion of edge motion has assumed that the material forming both sides of the edge was stretchable. When material is inelastic, both its left-hand and right-hand edges must be moved in tandem.

In particular, if the right-hand edge of a piece of inelastic material moves, its left-hand edge must also move. Figure 11 illustrates how this can lead to a cycle of dependencies. The plowing algorithm breaks this cycle by comparing the amount an edge is supposed to move with the motion distance already stored with the edge. If the stored motion distance is greater, the edge need not be moved a second time.

In cases where a layout rule violation exists in the original layout, an infinite loop is still possible. In Figure 11, for example, the distance  $r$  between edges  $f$  and  $e$  is less than  $s$ , the

minimum separation allowed. Edge  $e$  initially moves by distance  $d$ . Plowing should move all edges found in the shadow of  $f$  far enough away so as not to cause any rule violations with the newly moved  $f$ . Hence edge  $e$  would have to move by  $d+s-r$ , which is more than the motion distance stored with the edge. This leads to an infinite loop in which edge  $e$  is moved by an additional  $s-r$ .

Plowing avoids this sort of infinite loop by never moving a shadowed edge ( $e$ ) more than the edge causing the shadow ( $f$ ). This technique prevents infinite looping in over-constrained situations, but preserves existing layout rule violations.

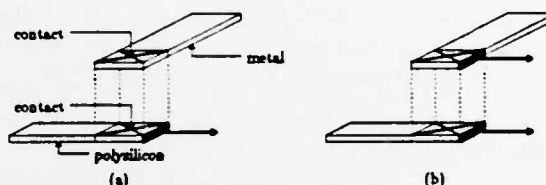


Figure 12. A contact is duplicated on each plane it connects. When an edge of a contact is moved on one plane, it is moved on all other planes as well.

## 5.3. Noninteracting planes

Section 4 explained that the order of vertical edges along a horizontal line is unchanged by plowing. Thus material being plowed can never slide over other material in its path. There are cases, however, where it is desirable that certain materials in a layout move independently. Metal, for example, does not interact with either polysilicon or diffusion except at contacts, so it should be able to slide over them.

To allow sliding, Magic segregates the mask information in a layout into a collection of non-interacting planes. Material in one plane is free to slide past material in any other plane. The nMOS technology, for example, has two planes: one to hold metal wires, and one to hold polysilicon, diffusion, and transistors.

The plowing algorithm operates on each plane independently. The only interaction between planes occurs at contacts, which are duplicated in each plane that they connect. When an edge of a contact is moved in one plane, the corresponding edge of the contact in all other planes is moved by the same amount, as illustrated in Figure 12. This also moves whatever the contact connects to in the other planes, thus preserving connectivity.

## 5.4. Jog control

Section 3.1 described how jog insertion was an automatic consequence of the rules plowing uses for finding edges to move. Plowing creates a jog whenever it moves only part of the boundary between two different types of material. Unfortunately, this often introduces a large number of jogs, which is bad both because it increases the size of the database needed to represent the layout, and because it may reduce fabricability. To control the number of jogs inserted by plowing, the user can specify a "jog horizon". Whenever an edge is about to be moved, plowing will attempt to extend it up and down to the nearest existing jog in each direction. If an existing jog is



found within the jog horizon of the corresponding endpoint of the edge, the existing jog is used; otherwise, the endpoint of the edge is used to form a new jog.

### 5.5. Subcells and hierarchy

One approach for plowing a hierarchical layout, such as that shown in Figure 13a, is to treat it as though it were non-hierarchical and propagate edge motions inside subcells. This might be workable when no subcell is used more than once. However, Magic instantiates subcells by reference, so a change in one instance of a subcell is reflected in all its other instances. Situations in which a subcell is used more than once can produce unsatisfiable sets of constraints, as Figure 13b illustrates.

Magic takes a simpler approach, which is to view subcells as black boxes to which connectivity must be maintained by plowing, but whose internal structure should not be modified. A benefit of Magic's approach is that plowing can be used to modify the placement of cells at the floor plan of a chip, since it only changes the location of subcells, not their contents.

When any mask geometry that abuts or overlaps a cell is moved, the entire cell must move by the same amount. Conversely, whenever a subcell moves, all mask geometry and

other subcells that abut or overlap it must also move by the same amount. The net effect is that a cell behaves like flypaper, causing all geometry over its area to "stick" to it and move as a whole when any part of it is required to move.

In addition to preserving connectivity with subcells, when plowing moves other geometry it must avoid introducing any layout rule violations with the geometry inside a subcell. One approach for dealing with this is to define a *protection frame* [Kell 82] for each cell, an outline around the cell into which no material may be plowed. Magic uses an extremely simple form of protection frame: it assumes that the cell contains all types of material right up to the border of its bounding box.

For example, in our nMOS rule set, the worst-case layout rule involving diffusion is the diffusion-diffusion spacing rule of 6 microns. An edge with diffusion to its left can be plowed to within 6 microns of a subcell before that subcell will itself have to move. The worst-case rule distance involving polysilicon is 8 microns, so polysilicon can only be plowed to within 8 microns of a subcell before the cell must move. Since the contents of subcells are considered unknown, the closest one subcell can be plowed to another before the other will have to move is the worst-case layout rule in the entire ruleset, which in our ruleset is 8 microns. Of course, if the user wishes to overlap two cells, he can still do that using other editing operations besides plowing.

### 6. Results and experience

Plowing has been implemented as part of the Magic VLSI layout system, which runs under the Berkeley 4.2 Unix operating system on either VAXes or Sun workstations. About 7500 lines of C code were required to implement all of the features described in this paper. The current version supports plowing only from left to right, but is currently being expanded to operate in all four directions. Table I gives measurements of the performance of the left-to-right version, using several examples taken from designs at Berkeley.

We have had no real user experience with the system yet, since it is just now becoming operational. However, the initial reaction from designers has been very positive. One recently-discovered problem has to do with line widths: the design rules only specify minimum widths for lines, so the current version of plowing will reduce the widths of lines that were initially wider than minimum width. This is unacceptable for many signals and especially for power and ground, so the plowing implementation is being modified to avoid reducing the widths of lines.

Although plowing is a rather tricky operation to implement correctly, it is very simple from the user's standpoint, and runs quickly enough to provide interactive response even for large cells. We hope that it will simplify the task of rearranging circuits topologically. If so, it will make it easier for designers to optimize their layouts, and will also help them to develop intuitions by allowing them to try out many alternative designs easily.

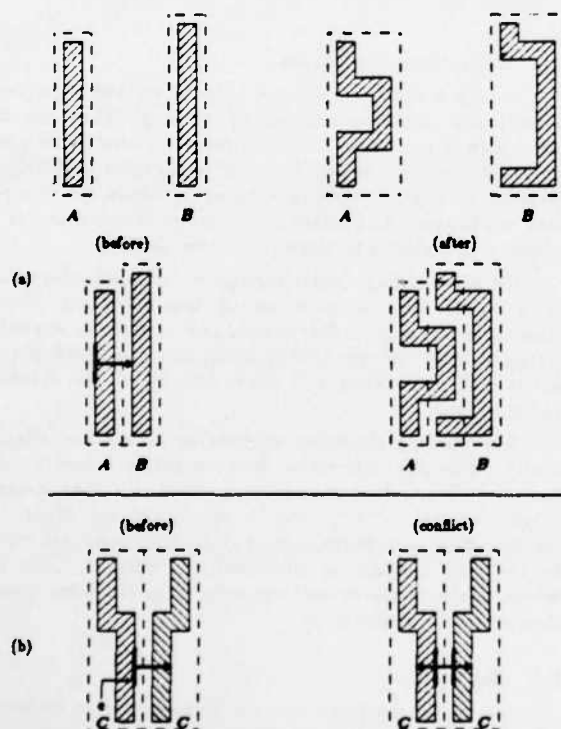


Figure 13. Plowing in the presence of hierarchy. (a) Plowing might treat hierarchy as though it were invisible to the user. Each of cells A and B would be modified. (b) Cell C is used twice, once flipped left-to-right and once in its normal orientation. Both uses refer to the same master definition of C. Moving edge e to the right is impossible, because it requires e to move to the left in order to keep out of its own path. The more edge e is moved to the right in the left-hand use, the worse the violation becomes.

Example	tiles	cells	edges	time
48-bit bus	101/480	0	382	2.5
ALU latch	430/472	0	785	3.0
Bus driver	848/1154	13	969	5.5

**Table I.** Plowing performance. The *tiles* column records the number of tiles of mask information in the cell being plowed, before/after plowing. The *cells* column records the number of subcells in the cell being plowed, and *edges* records the number of edges processed during plowing. The time is in seconds on a VAX-11/780.

## 7. Acknowledgements

Gordon Hamachi, Robert N. Mayo, and George Taylor all contributed to the discussions out of which the plowing algorithm arose. In addition to the above people, Randy Katz, Ken Keller, and Steve and Jean McGrogan all provided helpful comments on early drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

## 8. References

- [RBDD 83] Rosenberg, J., Boyer, D., Dallen, J., Daniel, S., Poirier, C., Poulton, J., Rogers, D., Weste, N. "A Vertically Integrated VLSI Design Environment." *Proceedings, 20th Design Automation Conference*, 1983, pp. 31-38.
- [Kell 82] Keller, K., Newton, A. "A Symbolic Design System for Integrated Circuits." *Proceedings of the 19th Design Automation Conference*, June 1982.
- [Oust 81] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI." *VLSI Design*, Vol. II, No. 4, Fourth Quarter 1981, pp. 34-38.
- [Oust 84] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." *IEEE Transactions on CAD/ICAS*, Vol 3, No. 1, January 1984, pp. 87-100.
- [OHMST 84] Ousterhout, J.K., Hamachi, G., Mayo, R.N., Scott, W.S., and Taylor, G.S. "The Magic VLSI Layout System." *Proceedings of the 21st Design Automation Conference*, June 1984.
- [TaOu 84] Taylor, G.S., and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." *Proceedings of the 21st Design Automation Conference*, June 1984.
- [West 81] Weste, Neil. "Virtual Grid Symbolic Layout." *Proceedings, 18th Design Automation Conference*, 1981, pp. 225-233.
- [Will 78] Williams, J. "STICKS- A Graphical Compiler for High Level LSI Design." *Proceedings of the 1978 NCC*, May 1978, pp. 289-295.

# COMPUTER-AIDED SYNTHESIS OF PLA-BASED FINITE STATE MACHINES

Giovanni De Micheli

Alberto Sangiovanni-Vincentelli

Department of EECS-University of California at Berkeley

Tiziano Villa

CSELT Torino (Italy) and Department of EECS-University of California at Berkeley

**Abstract:** We address the optimal logic design of PLA-based Finite State Machines (FSM). Techniques related to heuristic combinational logic minimization are used to determine optimal coding of the FSM internal states. We show that if appropriate Hamming-distance requirements among state codes are preserved, reduction of the combinational logic is guaranteed. A state encoding technique satisfying these requirements and based on graph embedding in squashed hypercubes is presented. Experimental results are reported.

## 1. INTRODUCTION

Sequential circuits play a major role in the control part of digital systems. We address the automated synthesis of sequential logic functions in a structured VLSI design methodology. We consider sequential logic functions implemented by synchronous deterministic Finite State Machines (FSM) consisting of two distinct components: a combinational circuit implemented by a Programmable Logic Array (PLA) and a memory implemented by Delay-type registers.

In particular we consider here the problem of assigning binary codes to the internal states of a Finite State Machine. The literature is rich of papers dealing with the state-assignment problem. Here we refer to the major approaches only. Armstrong [1] introduced a set of criteria for encoding states, aiming at the minimization of the number of gates used to implement the FSM and formulated the encoding problem as a graph embedding problem. Hartmanis [2], Stearns [3] and Karp [4] developed algebraic methods based on partition theory and on a reduced dependence criterion. Dolotta and McCluskey [5] suggested a "column-based" procedure to code states. Note that despite these efforts, to the best of our knowledge no tool for designing FSM is in use today for a time-effective state encoding of industrial digital controllers.

Armstrong's approach can in principle handle rather large machines, but it has three serious drawbacks. The first is related to the fact that the criteria suggested by Armstrong do not take into account the techniques of fast heuristic logic minimizers such as MINI [6], PRESTO [7], or ESPRESSO-II [8] in use today (Armstrong's paper appeared before the work on heuristic minimizers started). The second is that the state-assignment problem is transformed into a particular graph-embedding problem, which represents only partially the state coding problem, as shown in section 4. The third is that the graph embedding algorithm suggested by Armstrong was ineffective.

Our approach is based, as Armstrong's, on the use of distance relations among the codes of the internal states. In section 3 we show how the combinational logic can be reduced by requiring state codes to satisfy appropriate distances. Distance requirements are determined by predicting the effects of heuristic minimization of the combinational logic related to a symbolic description of the FSM, and are represented by a graph. In particular it is shown that a convenient reduction of the combinational logic is obtained if the distance between some state codes is large enough and appropriate states have adjacent codes.

In section 4 we consider the problem of assigning codes which satisfy the distance relations. Adjacent code assignment can be seen as an embedding of an adjacency graph into a boolean hypercube. Armstrong [1] and Saucier [9] represented the state assignment problem as a subgraph isomorphism problem, where a one-to-one relation (coding) is sought between the set of the states (vertices of the adjacency graph) and a subset of the boolean hypercube vertices (codes).

Note that even questioning the existence of a subgraph isomorphism is a hard problem: in particular it was shown to belong to the class of NP-complete problems [10]. Since such an isomorphism may not exist, Armstrong and Saucier relaxed some adjacency requirements and proposed heuristic techniques to embed a subgraph of the adjacency graph into the boolean hypercube. Note that a distance-preserving embedding is not even guaranteed by augmenting the dimensions of the hypercube, i.e. increasing the length of the state codes.

Our approach exploits the use of *don't care* conditions in state codes. In particular every state is coded by associating each vertex of the adjacency graph to a subcube of the boolean hypercube. This is equivalent to embed the adjacency graph into a squashed hypercube, i.e. a hypercube having appropriate faces squeezed into vertices [11]. Note that most of the state assignment techniques presented in the literature obtained a state coding using the minimum number of bits, because it was important to minimize the number of memory elements due to their cost. On the other hand, the area taken by the PLA is the

major concern in a VLSI circuit implementation of a Finite State Machine. Minimal area PLA implementations of the FSM combinational component can be obtained by using non-minimal-length state codings i.e. fewer product-terms are often required to implement a logic function at the expense of an increased number of input/output columns. Therefore we allow non-minimal-length state codings when leading to minimal area PLAs. In this case, state coding corresponds to an embedding into a squashed hypercube of variable dimension. However bounds on code-length can be enforced when required by a particular implementation.

## 2. FINITE STATE MACHINE REPRESENTATION

Different functional FSM representations are commonly used. Most state-assignment techniques reported in the literature are based on a state-table representation, though it can be cumbersome for large uncompletely-specified machines. For this reason designers describe the machine functionality by means of flow-charts or Hardware Description Languages (HDL). Unfortunately these descriptions are not well-suited to support machine optimization techniques. For these reasons we represent the FSM functionality by means of a symbolic cover. The concept of symbolic cover is a generalization of the logic cover representation of combinational-logic functions [6]. Symbolic covers can be obtained from flow-charts, HDL or state tables in a straightforward way.

A symbolic cover is a set of primitive elements called symbolic implicants. A symbolic implicant (denoted here by a capital letter e.g.  $A = \{i_A, s_A, s'_A, o_A\}$ ) is a set of two input and two output character strings. The two input strings represent a binary-valued representation of a primary input ( $i_A$ ) and a symbolic representation of a present state ( $s_A$ ). The two output strings represent the corresponding symbolic representation of the next-state ( $s'_A$ ) and a binary-valued representation of the primary outputs ( $o_A$ ). Note that we consider in this paper the problem of assigning binary codes to the FSM internal states only. Therefore we assume that  $i_A$  and  $o_A$  are already coded into binary strings. However  $i_A$  and  $o_A$  might describe symbolic inputs and outputs in a more general framework, where primary input and output coding is also considered. We represent binary valued variables by the symbols "1", "0" and "c", where "c" represents a *don't care* condition. States are represented symbolically by a character mnemonic string.

**Example:** Consider the traffic-light controller presented in [12]. The following is a symbolic implicant:

11\*.HG.HY.10010

showing the a "1" in the first two primary-input lines maps state "HG" into state "HY" and asserts output 10010. The symbolic cover is the collection of the symbolic implicants representing the state transitions:

00\*.HG.HG.00010  
00\*.HG.HG.00010  
11\*.HG.HY.10010  
00\*.HY.HY.00110  
01.HY.FG.10110  
10\*.FG.FG.01000  
00\*.FG.FY.11000  
01\*.FG.FY.11000  
00\*.FY.FY.01001  
01.FY.HG.11001

Note that a symbolic cover is a logic cover of a multiple-valued logic function [6] [13], where each state takes a different logic level and is represented by a character string. A symbolic implicant having  $n$  (m) primary input (output) bits can be seen as a  $(n+1)$ -input,  $(m+1)$ -output multiple-valued logic implicant. Definitions and properties of multiple-valued-logic covers carry over to symbolic covers as well [13].

The motivation for using a symbolic cover relies on the following points.

i) properties and operations on symbolic covers can be exploited and related to heuristic minimization algorithms for binary-valued logic functions [8],[7],[6].

ii) any logic cover of the combinational component of a FSM obtained by assigning disjoint codes to each state can be seen as a symbolic cover. Hence the technique we present can be interfaced to several FSM automated design tools, in order to implement the machine aiming specifically to a PLA-based implementation in a minimal area.

The state assignment problem consists of determining a coding map  $c(\cdot)$  which transforms the state symbols into strings of binary digits. This is equivalent to transforming the symbolic cover into a binary-valued logic cover of the combinational component. Note that in general don't care coordinates are used in state codes, and therefore every state is assigned to a subcube of the boolean hypercube. However a coding map is implementable only if the states are assigned to non-overlapping regions of the boolean hypercube.

State coding affects substantially the complexity of the combinational component of a FSM, because minimal binary-valued logic covers [6] corresponding to different coding maps have different cardinalities. We consider first coding maps with no code-length bounds. The unconstrained optimum state assignment problem can be stated as follows:

Find an implementable coding map  $c(\cdot)$  that minimizes the cardinality of the minimal logic cover of the FSM combinational component.

This is a formidable task, because it involves the search for all the minimal covers related to all possible codings. We therefore concentrate on a simpler problem and we relate optimal state coding to heuristic minimization of the logic cover [6] [8]. In particular we look for an implementable coding map which leads to a minimal logic cover having significantly fewer implicants than the original symbolic cover. Similarly a constrained state assignment problem can be defined by restricting the search to codings of bounded length.

### 3. CODE DISTANCES AND COMBINATIONAL LOGIC MINIMIZATION

We investigate in this section the relations between state assignment and the complexity of the related implementation of the combinational part of a FSM. In particular a set of rules can be obtained to determine constraints on state code distances, so that either the cardinality or the number of literals of the logic cover (or both) can be reduced. However we report here on the two major rules only.

We call cube any string of characters from the set  $\{0,1,*\}$ . We refer the reader to [6] for definitions of cover ( $\supset$ ), union ( $\cup$ ), sharp ( $\cdot$ ) and intersection ( $\cap$ ) between cubes. The distance  $D(a, b)$  between two cubes  $a$  and  $b$  of equal length is the number of positions in which they differ. The Hamming distance  $H(a, b)$  between two cubes  $a$  and  $b$  of equal length is the number of positions in which they differ and both entries are cares. Note that if  $s_1$  and  $s_2$  are two different state symbols, an implementable coding is such that  $H(c(s_1), c(s_2)) > 0$ . We define two state codes to be adjacent, if their Hamming distance is one, because they are adjacent vertices of a squashed cube representation.

The basic strategy for obtaining a set of relations among state codes is the following. All pairs of symbolic implicants ( $A, B$ ) are examined and code distance requirements are enforced according to the following rules. When Rule 1 applies, two symbolic implicants can be coded and merged into one binary-valued logical implicant and the cover cardinality be reduced. Therefore Rule 1 is considered a "strong rule" and it is highly desirable that the related code distance requirements are satisfied. Rule 2 allows to reduce the number of literals and is considered a "weak rule" compared to Rule 1, because a reduction in size of the PLA is considered more desirable than a reduction of its complexity.

Let  $A = \{i_A, s_A, s'_A, o_A\}$  be a symbolic implicant of the machine cover. We define  $S(A)$  the set of states which are mapped by any input representation  $i \in i_A$  either into a next-state different from  $s'_A$  or into an output representation not covered by  $o_A$  or both.

Rule 1: Let  $A = \{i_A, s_A, s'_A, o_A\}$ ,  $B = \{i_B, s_B, s'_B, o_B\}$  be two symbolic implicants such that:  $i_A \supset i_B$  and  $o_A \supset o_B$ . Then:

$$c(s'_A) \supset c(s'_B).$$

and:

$$H(c(s_A) \cup c(s_B), c(s_Q)) > 0 \quad \forall s_Q \in S(A).$$

Rationale:  $A$  and  $B$  can be coded and merged into only one logic implicant, namely:

$$\{i_A, c(s_A) \cup c(s_B), c(s'_A), o_A\}$$

Rule 1 requires two different conditions on state codes: i) a covering relation between cubes  $c(s'_A)$  and  $c(s'_B)$  considered as output parts of binary-valued implicants; ii) a distance relation which keeps state codes  $c(s_A)$  and  $c(s_B)$  far from the codes of the states in  $S(A)$ .

Remark: If  $s'_A = s'_B$  the covering requirement is automatically satisfied. Moreover if only completely specified codes are used (i.e. no don't care conditions are used in state codes), then  $D(c(s_A), c(s_B)) = 1$  implies that  $H(c(s_A) \cup c(s_B), c(s_Q)) > 0 \quad \forall s_Q \neq s_A$  and  $s_Q \neq s_B$ . This requirement is equivalent to the column adjacency rule stated by Armstrong in [1], when  $i_A = i_B$  and  $o_A = o_B$ . Note that Rule 1 is far more general than Armstrong's rule.

Let  $A = \{i_A, s_A, s'_A, o_A\}$  and  $B = \{i_B, s_B, s'_B, o_B\}$  be two symbolic implicants such that:  $s_A = s_B$  and  $o_A = o_B$ . We define  $I(AB)$  the set of input representations  $i \in i_A \cup i_B$  which map state  $s_A$  either into a next-state different from  $s'_A$  or  $s'_B$  or into an output representation not covered by  $o_A$  or both.

Rule 2: Let  $A = \{i_A, s_A, s'_A, o_A\}$  and  $B = \{i_B, s_B, s'_B, o_B\}$  be two symbolic implicants such that:  $s_A = s_B$  and  $o_A = o_B$ . If  $I(AB) = \emptyset$ , then:

$$H(c(s'_A), c(s'_B)) = 1.$$

Rationale: the corresponding logical implicants can be reshaped [6] as:

$$\{i_A \cup i_B, c(s_A), \tilde{c}(s'_A), o_A\} \\ \{i_A, c(s_B), c(s'_B) - c(s'_A), \emptyset\}$$

where  $\emptyset$  is a string of "0"s and where without loss of generality  $c(s'_A)$  and  $c(s'_B)$  are obtained by assigning cares to the don't care entries of the next-state codes, so that  $D(c(s'_A), c(s'_B)) = 1$  and the "1" count in  $c(s'_B)$  is larger than in  $c(s'_A)$ . Note that the second logical implicant obtained by Rule 2 has always only one care in the output part. Hence it may be covered by some other implicants of the logical cover. Therefore when Rule 2 applies the number of literals and possibly the logical cover cardinality are reduced.

Rule 2 requires an adjacency relation between cubes  $c(s'_A)$  and  $c(s'_B)$ .

Remark: If  $D(i_A, i_B) = 1$ , then  $I(AB) = \emptyset$ . Therefore, if we restrict our attention to completely specified codes only,  $D(c(s'_A), c(s'_B)) = 1$  implies that  $H(c(s'_A), c(s'_B)) = 1$ . This condition is equivalent to the row adjacency rule presented by Armstrong in [1].

More complex rules can be derived by considering other relations between symbolic implicants. In particular, Rule 2 can be generalized to the case in which  $o_A \supset o_B$  and Rule 1 be modified to the case in which  $H(o_A, o_B) = 1$ .

### 4. STATE ENCODING STRATEGIES

The rules stated in Section 3 give rise to relations among state codes which can be grouped as follows:

- 1) code adjacency ( $H(c(s_A), c(s_B)) = 1$ );
- 2) code covering, i.e. requiring a next-state code to cover another next-state code ( $c(s'_A) \supset c(s'_B)$ );
- 3) code distance, i.e. requiring the code of one state, say  $s_Q$ , to be far enough from the union of the codes of a pair of states  $s_A$  and  $s_B$  ( $H(c(s_A) \cup c(s_B), c(s_Q)) > 0$ ).

Relations 1) and 2) are represented by a mixed weighted graph,  $G(V, E, W(E))$ , where the set of nodes  $V$  is in one-to-one correspondence with the set of states, and  $E$  consists of a set of directed and undirected edges. The undirected edges are related to the adjacency relations, i.e.  $\{u, v\} \in E$  if  $H(c(s_u), c(s_v)) = 1$ ; and the directed edges are related to covering relations, i.e.  $(u, v) \in E$  if  $H(c(s_u), c(s_v)) = 1$  and  $c(s_u) \supset c(s_v)$ . Weights are defined according to the number of times the same distance requirement occurs and to the related rule. Code distance requirements are represented by a list structure. In particular  $H(c(s_A) \cup c(s_B), c(s_Q)) > 0$  is represented by  $s_Q$  pointing to the pair  $s_A$  and  $s_B$  in the list.

The problem of finding a state coding which satisfies the rules given in Section 3 can now be seen as a graph embedding problem. Let  $N$  be the dimension of a boolean hypercube  $B$ , representing the possible codes. Let  $P(B)$  be the set of all subcubes contained in  $B$ . We have to determine the dimension  $N$  and an injective function  $c: V \rightarrow P(B)$  such that the relations induced by the rules presented in Section 3 are satisfied. The adjacency relations are satisfied if:

$$d_G(u_i, u_j) \geq H(c(u_i), c(u_j)) \quad \forall u_i, u_j, u_i \neq u_j \in V$$

where  $d_G(u_i, u_j)$  is the length of the shortest path in the graph between  $u_i$  and  $u_j$  and  $H(c(u_i), c(u_j))$  denotes the Hamming distance of the subcubes  $c(u_i)$  and  $c(u_j)$ . Geometrically, we would like to determine, if possible, an isomorphism between the graph and a squashed hypercube, i.e. an hypercube  $B$  in which some elements of  $P(B)$  collapse into a vertex. It can be proven that there always exists an integer  $N$  such that an injective map  $c: B \rightarrow P(B)$  satisfying the above relations can be found. However, it is important not only to reduce the number of product terms of the FSM combinational component, but also to keep  $N$  as small as possible because  $N$  is proportional to the number of columns required by a PLA implementation.

Three optimization strategies can be followed:

- 1) Set  $N$  to a fixed value and find  $c(\cdot)$  such that the number of code constraints violated by the encoding is minimized;
- 2) Find the smallest  $N$  such that all the rules are satisfied;



### 3) Trade-off $N$ and the number of constraints violated.

Strategy 1 is close to the one followed by Armstrong where  $N = \lceil \log_2 |V| \rceil$ , i.e. the minimum number of bits needed to encode the states. Strategy 3 is the most desirable but obviously the most difficult to implement. We decided to implement strategy 2 as an intermediate step towards strategy 3. A first theoretical question to ask, when implementing strategy 2, is whether a bound on  $N$  can be found.

If we require that:

$$d_G(u_i, u_j) = H(c(u_i), c(u_j)) \quad \forall u_i, u_j, u_i \neq u_j \in V$$

we have an isometric embedding of a graph into a squashed hypercube. Graham showed in [11] that any graph  $G(V, E)$  can be embedded into an hypercube of dimension  $N(G) = \lceil |V| - 1 \rceil \text{diam}(G)$  where  $\text{diam}(G)$  is the diameter of  $G$ , and conjectured that the bound can be lowered to  $N(G) = |V| - 1$ . The conjecture can be proven true for graphs belonging to some special classes, e.g. complete graphs. The graph embedding problem arising from our formulation is a distance-bounded graph embedding. It can be reduced to an isometric embedding into a squashed hypercube by appending appropriate edges to  $G$ . Therefore there always exists a coding map  $c(\cdot)$  satisfying the given requirements having  $N(G) = |V| - 1$ .

We present in Fig. 1 the flow-chart of a heuristic algorithm for distance-bounded graph embedding, which reminds of the procedure presented in [14] for isometric embedding. The algorithm tries to minimize  $N$  and is constructs a coding using  $N \leq |V| - 1$  bits. Note that this is a worst-case upper bound and that the computed codes are much shorter than  $|V| - 1$  in many practical cases.

The algorithm applies to connected graphs. If  $G(V, E, W(E))$  is disconnected, its connected components are determined first and the different groups of codes are packed together at the end. We deal here with a connected graph for the sake of simplicity.

The algorithm visits each node of the graph  $v_k$ ,  $k=1, \dots, |V|$  and at the  $k$ -th step constructs a partial encoding of length  $N(K)$  for  $v_k$ . It appends one bit to the codes of the nodes  $v_i$ ,  $i=1, \dots, k$  only if the code length must be increased, as shown in Fig. 1. A degree of freedom of our procedure is the order of the selected nodes. We choose as first node the one which corresponds to the state with maximum number of occurrences as next-state in the symbolic cover. We map it into the origin of the coordinates of the hypercube, to maximize the occurrence of "0"s in the output part of the coded implicants. The node selected at the  $k$ -th step,  $v_k$ , is adjacent to a coded vertex (i.e. adjacent to  $v_i, i < k$ ) and has the maximum number of uncoded adjacent nodes. The rationale is that such a node has more constraints to satisfy and so it deserves higher priority in the space occupation on the hypercube.

At step  $k$  node  $v_k$  is coded as follows. Assume we have assigned partial codings of code length  $N(K-1)$  to  $v_i$ ,  $i=1, \dots, k-1$  so that:

$$H(c(u_i), c(u_j)) = 1$$

for all adjacent coded pairs  $u_i$  and  $u_j$ . Then we search for an implementable coding  $c(v_k)$  of the same length with the property that:

$$H(c(v_k), c(u_j)) = 1$$

for all adjacent coded pairs  $u_i$  and  $v_j$ , under the constraint:

$$H(c(v_k), c(u_i)) \cup c(u_i) > 0$$

for all node pairs  $u_i, u_j$  in the list pointed by  $v_k$  and coded before step  $k$ . An exhaustive search of a feasible code would require  $3^{N(K-1)}$  trials. Therefore we test only a subset of the possible trials, which are called "slight modifications" of the coding of the vertices adjacent to  $v_k$ . A slight modification is obtained by complementing one care bit ("1" or "0") of the code of a vertex adjacent to  $v_k$ . There are at most  $|V|$  such trials.

It is possible that no implementable coding for  $v_k$  can be obtained by slight modifications. In this case the algorithm constructs the code of  $v_k$  by appending a "1" to the string of bits obtained from the logical union of the codes of all the adjacent vertices and by appending a "1" or "0" or "0" to the code of each vertex  $v_i$  coded before step  $k$ . In this way, we can always satisfy the distance requirements, but unfortunately at the expense of an increase in the code length. However, the algorithm will construct a valid encoding for  $G$  of length bounded by  $1 + \sum_{i=1}^{|V|} 1 = |V| - 1$ . Its computational complexity is  $O(|V|^3)$  in the worst case.

**Remark:** Since a bound on the code length can be obtained by bounding the number of vertices in each connected component of  $G$ , we can partition the graph into components of bounded size by removing a subset of edges. Edge weights can be used to determine the optimal graph decomposition.

### 5. EXPERIMENTAL RESULTS AND CONCLUDING REMARKS

The algorithm has been implemented by an interactive computer program. The program reads the symbolic description of a FSM, generates the distance requirements and determines state codes. The program has been tested on a set of industrial Finite State Machines. Results are reported in Table 1 and show that the algorithm is effective in generating state codes leading to a FSM implementation with a reduced number of product-terms in the combinational component. Execution times are in the order of some seconds on a IBM 3081 computer.

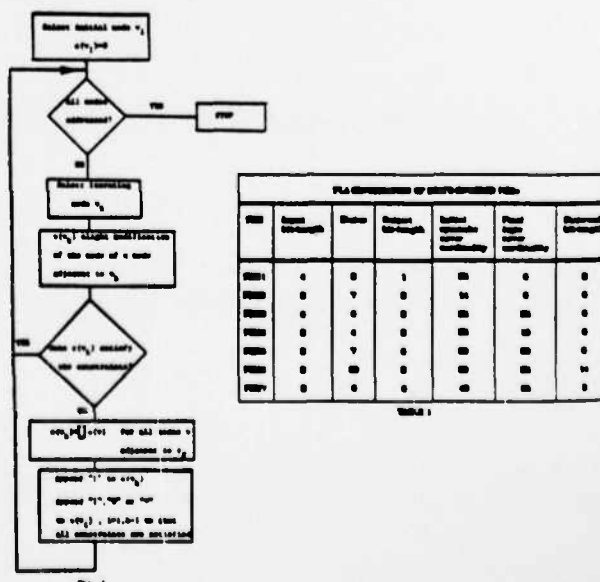
A new approach based on multiple-valued logic minimization is being currently pursued in collaboration with Dr. Brayton of IBM. Preliminary experimental results show that this method can be considered a break-through in FSM synthesis.

### 6. ACKNOWLEDGMENTS

This research has been sponsored by IBM, NSF under subcontract # 392741C-1, DARPA under contract # 25978 and CSELT, Italy.

### REFERENCES

- [1] D.B. Armstrong "A Programmed Algorithm for Assigning Internal Codes to Sequential Machines" *IRE Trans. Elect. Comp.* vol EC-11 pp 466-472, aug 1962.
- [2] J. Hartmanis "On the State Assignment Problem for Sequential Machines I" *IRE Trans. Elect. Comp.* vol EC-10 pp 157-166, jun 1961.
- [3] R.E. Stearns and J. Hartmanis "On the State Assignment Problem for Sequential Machines II" *IRE Trans. Elect. Comp.* vol EC-10 pp 563-603, dec 1961.
- [4] R. Karp "Some Techniques for State Assignment for Synchronous Sequential Machines" *IEEE Trans. Elect. Comp.* vol EC-13 pp 507-516, oct 1964.
- [5] T.A. Dolotta and E.G. McCluskey "The coding of internal states of sequential machines" *IEEE Trans. Elect. Comp.* vol EC-13 pp 549-562, oct 1964.
- [6] S.J. Hong, R.G. Cain and D.L. Ostapko "MINI: A Heuristic Approach for Logic Minimization" *IBM Jour. of Res. and Dev.* vol 16, pp 443-458, sep 1974.
- [7] D. Brown "A State Machine Synthesizer" *Proc. IBA Des. Aut. Conf.*, Nashville, jun 1961.
- [8] R. Brayton, G.D. Hachtel, C. McMullen and A.L. Sangiovanni-Vincentelli "ESPRESSO-II: A New PLA Logic Minimization Program" in preparation.
- [9] G. Seucier "State Assignment of Asynchronous Sequential Machines Using Graph Techniques" *IEEE Trans. Comp.* vol C-21 pp 282-288, mar 1972.
- [10] M.R. Garey and D.S. Johnson "Computers and Intractability" W.H. Freeman and Company, San Francisco 1978.
- [11] R.L. Graham and H.O. Pollak "On Embedding Graphs in Squashed Cubes" *Graph Theory and Applications* Lecture notes in mathematics, no. 303, Springer Verlag 1972.
- [12] C. Mead and L. Conway "Introduction to VLSI Systems" Addison Wesley, 1981.
- [13] S.Y.H. Su and P.T. Cheung "Computer Minimization of Multi-Valued Switching Functions" *IEEE Trans. on Comp.* vol 21, 1972, pp 995-1003.
- [14] R.L. Graham and H.O. Pollak "On The Addressing Problem for Loop Switching" *Bell Syst. Tech. Jour.* vol 50 No 6, pp 2489-2519, oct 1971.



FSM	Nodes	Edges	Output	Input	Output	Input	Output	Input
FSM1	4	8	1	0	0	0	0	0
FSM2	4	8	1	0	0	0	0	0
FSM3	4	8	1	0	0	0	0	0
FSM4	4	8	1	0	0	0	0	0
FSM5	4	8	1	0	0	0	0	0
FSM6	4	8	1	0	0	0	0	0

# Relaxation-Based Electrical Simulation

ARTHUR RICHARD NEWTON, MEMBER, IEEE, AND ALBERTO L. SANGIOVANNI-VINCENTELLI,  
FELLOW, IEEE

**Abstract**—Circuit simulation programs have proven to be most important computer-aided design tools for the analysis of the electrical performance of integrated circuits. One of the most common analyses performed by circuit simulators and the most expensive in terms of computer time is nonlinear time-domain transient analysis. Conventional circuit simulators were designed initially for the cost-effective analysis of circuits containing a few hundred transistors or less. Because of the need to verify the performance of larger circuits, many users have successfully simulated circuits containing thousands of transistors despite the cost.

Recently, a new class of algorithms has been applied to the electrical IC simulation problem. New simulators using these methods provide accurate waveform information with up to two orders of magnitude speed improvement for large circuits. These programs use *relaxation* methods for the solution of the set of ordinary differential equations, which describe the circuit under analysis, rather than the direct sparse-matrix methods on which standard circuit simulators are based.

In this paper, the techniques used in relaxation-based electrical simulation are presented in a rigorous and unified framework, and the numerical properties of the various methods are explored. Both the advantages and the limitations of these techniques for the analysis of large IC's are described.

## I. INTRODUCTION

CIRCUIT simulation programs, such as SPICE2 [1] and CASTAP [2], have proven to be most important computer-aided design tools for the analysis of the electrical performance of integrated circuits (IC's). These programs can perform a variety of analyses, including dc, ac, and time-domain transient analysis of circuits containing a wide range of nonlinear active circuit devices such as MOSFET's and bipolar junction transistors [3].

One of the most common analyses performed by circuit simulators and the most expensive in terms of computer time is nonlinear time-domain transient analysis. By performing this analysis, precise electrical waveform information can be obtained if the device models and parasitics of the circuit are characterized accurately. However, conventional circuit simulators were designed initially for the cost-effective analysis of circuits containing a few hundred transistors or less. Because of the need to verify the performance of larger circuits, many users have successfully simulated circuits containing thousands of transistors despite the cost. For example, a 700 MOSFET circuit, analyzed for 4  $\mu$ s of simulated time with an average 2-ns time step, takes approximately 4 CPU hours on a VAX

11/780 VMS computer with floating-point accelerator hardware.

Gate-level logic simulators (e.g., [4], [5]) and switch-level simulators [6]–[8] can verify circuit function and provide first-order timing information more than three orders of magnitude faster than a detailed circuit simulator. However, to verify circuit performance for critical paths, memory design, and analog circuit blocks, it is often essential to perform accurate electrical simulation. In some companies the simulation of circuits containing many thousands of devices is performed routinely and at great expense. In recent years, considerable effort has been focussed on techniques for improving the speed of time-domain electrical analysis while maintaining acceptable waveform accuracy.

A number of approaches have been used to improve the performance of conventional circuit simulators for the analysis of large circuits. The time required to evaluate complex device model equations has been reduced using table-lookup models [9]–[13]. Techniques based on special-purpose microcode have been investigated for reducing the time required to solve sparse linear systems arising from the linearization of the circuit equations [14]. Node-tearing techniques have also been used to exploit circuit regularity by bypassing the solution of subcircuits whose state is not changing [15], [16] and to exploit the vector-processing capabilities of high-performance computers such as the CRAY-1 [17]. In all cases, the overall speed improvement of the simulation has been at most an order of magnitude, for practical circuits.

Recently, a new class of algorithms has been applied to the electrical IC simulation problem. New simulators using these methods provide as accurate, or more accurate, waveforms than standard circuit simulators such as SPICE2 or ASTAP with up to two orders of magnitude speed improvement for large circuits. These simulators have been used for the analysis of both digital and analog MOS IC's. They use *relaxation* methods for the solution of the set of ordinary differential equations, (ODE's) which describe the circuit under analysis, rather than the direct sparse-matrix methods on which standard circuit simulators are based.

A broad survey of decomposition techniques for the simulation of large-scale integrated circuits can be found in [18]. In this paper, the techniques used in relaxation-based electrical simulation are presented in a rigorous and unified framework and the numerical properties of the various methods are explored. Both the advantages and the limitations of these techniques for the analysis of large IC's are

Manuscript received May 25, 1983. This work was supported in part by DARPA under Contract N00030-K-0251, by JSEP under Contract AFOSR-F49620-79C0178, by ARO under Contract DAAG29-81-K-0021, and by NSF under Contract ECS-7913148.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

described. In Section II, some of the fundamental problems associated with conventional circuit simulation algorithms as circuit size increases are exposed and the mathematical basis for the relaxation approach is introduced. In Section III, the special relaxation methods called *timing simulation* algorithms are described and their numerical properties are investigated. In Section IV, *iterated timing analysis*, which applies relaxation techniques at the nonlinear equation level [19], is described briefly and its convergence properties are proven. The *waveform relaxation* method [20], [21], which applies relaxation techniques at the differential equation level, is presented in Section V, and various techniques which can be used to improve its performance for electrical simulation are described. Concluding remarks and areas requiring further research are presented in Section VI.

## II. CIRCUIT EQUATION FORMULATION AND STANDARD RELAXATION TECHNIQUES

### A. Equation Formulation

Before the techniques used in relaxation-based simulation are presented, the particular electrical simulation problem to be solved must be defined. Although relaxation-based methods can be used with a variety of technologies (e.g., [23]), they are particularly suited to the analysis of large MOS digital IC's, as will become clear later. Thus to help clarify the presentation, the following simplifying assumptions are made:

- All resistive elements, including active devices, are characterized by constitutive equations where voltages are the controlling variables and currents are the controlled variables.
- All energy storage elements are two-terminal, possibly nonlinear, voltage-controlled capacitors.
- All independent voltage sources have one terminal connected to a ground or can be transformed into independent current sources with the use of the Norton transformation.

Under these assumptions, the circuit equations can be formulated in terms of a nodal analysis that yields  $N$  equations in  $N$  unknown node voltages [24], where there are  $N+1$  nodes in the circuit and node  $N+1$  is the reference node, or ground.

An important assumption required by relaxation-based electrical simulators is that a two-terminal capacitor be connected from each node of the circuit to the reference node. This assumption is satisfied by circuits where lumped parasitic capacitances are present between circuit interconnect and ground or on the terminals of active circuit elements.

Under these assumptions, the nodal equations can be written in the form

$$C(v(t), u(t)) \dot{v}(t) = -f(v(t), u(t)), \quad 0 \leq t \leq T$$

$$v(0) = V \quad (1)$$

where  $v(t) \in \mathbb{R}^n$  is the vector of node voltages at time  $t$ ;

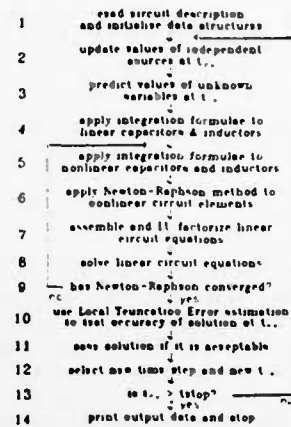


Fig. 1. Circuit simulator flow diagram for transient analysis.

$\dot{v}(t) \in \mathbb{R}^n$  is the vector of time derivatives of  $v(t)$ ;  $u(t) \in \mathbb{R}^n$  is the input vector at time  $t$ ,  $C(\cdot): \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$  represents the nodal capacitance matrix,  $f: \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ , and

$$f(v(t), u(t)) = [f_1(v(t), u(t)), f_2(v(t), u(t)), \dots, f_N(v(t), u(t))]^T$$

where  $f_i(v(t), u(t))$  is the sum of the currents charging the capacitors connected to node  $i$ . In the following sections (1) will be referred to in a simplified form where the time dependencies are expressed implicitly, i.e.,

$$C(v, u) \dot{v} = -f(v, u). \quad (2)$$

### B. Standard Circuit Simulation

A simplified flow diagram for the solution of these equations by a conventional circuit simulator is shown in Fig. 1. Once the circuit description has been read by the program and the data structures required for simulation have been assembled, the main analysis loop (Steps (2)–(13)) is entered.

At each new analysis time point,  $t_{n+1}$ , the information from previous time points is used to predict the solution at  $t_{n+1}$ . Stiffly stable integration formulas, such as Backward Euler (BE), the Trapezoidal Rule (TR), or Gear's Variable-Order Method (GE), with variable time steps, are used to discretize (1) at Steps (4) and (5) [3]. This process yields a set of nonlinear, algebraic difference equations of the form

$$g(x) = 0 \quad (3)$$

where  $x \in \mathbb{R}^N$  is the vector of node voltages at time  $t_{n+1}$ .

These equations are solved using a damped Newton-Raphson algorithm to yield a set of sparse linear equations of the form

$$Ax = b \quad (4)$$

where  $A \in \mathbb{R}^{N \times N}$  is a matrix related to the Jacobian of  $g$  and  $b \in \mathbb{R}^N$  [3]. Typically, less than 2 percent of the entries of  $A$  are nonzero for  $N > 500$ . These equations are then solved using direct methods, such as sparse LU decomposition or Gaussian Elimination, Steps (7) and (8).



Steps (5)–(9) are repeated until the Newton–Raphson process converges or the upper bound on the number of iterations is reached. The program then decides whether to accept the solution, based on its estimate of local truncation error (LTE) and the number of Newton–Raphson iterations required in Steps (5)–(9). A new time step is computed, and Steps (2)–(13) are repeated until the simulation is complete [3].

This procedure has proven to be reliable and accurate. For large circuits, the process can take a considerable amount of computer time, as illustrated in Section I. The majority of the time spent in Steps (2)–(13) can be lumped into two categories: the time required to solve the system of sparse linear equations, SOLVE (Steps (7) and (8)), and the time required to form the entries of  $A$  and  $b$  in (4), FORM (Steps (5) and (6)).

Fig. 2 shows the amount of CPU time required to perform a transient analysis of a set of typical circuits of increasing size. For this example, the number of circuit nodes  $N$  is used as a measure of circuit size. The time required for equation preprocessing is not included here; only time involved in the actual time-domain transient portion of the simulation is shown. A simple RC circuit was chosen for this example to emphasize the increasing cost of matrix solution time. The example was constructed by calling an increasing number of cells, in an hierarchical manner, each with the same matrix structure and an average number of fanouts between 2.5 and 3. This approach preserved the observed properties of most real circuits while providing a uniform technique for increasing circuit size.

As can be seen in Fig. 2 for small circuits ( $N < 20$ ), the majority of the solution time is spent performing FORM. However, when the size of the circuit grows, an increasing percentage of the time is spent in the SOLVE phase. While the actual percentages may vary depending on the circuit under analysis, the complexity of the nonlinear device models used by the program, and the computer on which the simulator is running, this trend is true for all standard circuit simulators running on conventional computers. For MOS circuits analyzed on a VAX11-780 UNIX computer, the crossover point is at around 500 nodes. The time spent in the equation solution phase has been measured to grow as  $O(N^\beta)$ , where  $1.1 < \beta < 1.5$ . In particular, for large circuits  $\beta$  has been found to depend on the difference between the time required to perform arithmetic operations and the memory bandwidth of the computer. On the other hand, the time required for FORM grows linearly with the number of circuit elements and, therefore, with the number of circuit equations for typical circuits. The time spent in the load phase can be reduced by simplifying the device model equations, using table look-up models [9]–[13], or providing special-purpose instructions to update  $A$  and  $b$  [14].

For most circuits the fraction of nodes which are changing their voltage value at a given point in time decreases as the circuit size increases. For circuits containing over 500 MOSFETS, fewer than 20 percent of the node voltages

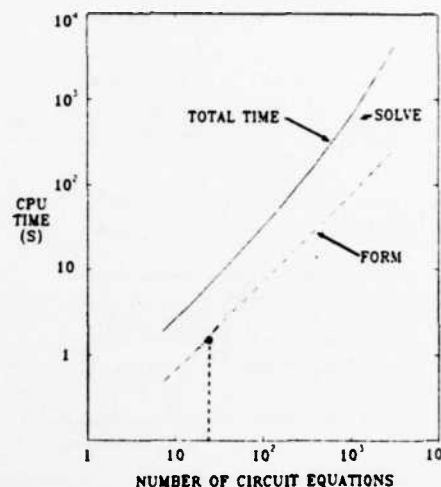


Fig. 2. Transient analysis time for circuits of increasing size.

change significantly over a simulation time step. Only the circuit equations representing these active nodes must be solved at any time. Circuit simulators exploit this *time sparsity* or *latency* by using device-level [1] or block-level [16], [25], [17] bypass schemes. In a device-level bypass scheme, if the terminal voltages and branch currents of a circuit element did not change significantly in the previous Newton–Raphson iteration, its contributions to  $A$  and  $b$  in (4) are not reevaluated, and the values computed during the previous iteration are used. In block-level bypass, both the matrix element evaluation and the node solution steps are bypassed for each block of inactive connected circuit elements. While the aforementioned techniques do reduce the total execution time for conventional circuit simulators, the savings are often not sufficient for the cost-effective electrical simulation of LSI circuits.

### C. Linear Relaxation Methods

Relaxation methods can be used for the solution of (1) in a number of ways. In all cases, their principal advantages stem from the fact that they do not require the direct solution of a large system of linear equations and from the fact that they permit the simulator to exploit latency efficiently.

Relaxation methods can be applied at different stages in the solution of (1), as illustrated in Fig. 3. The two most common methods used in electrical simulation are the Gauss–Jacobi method and the Gauss–Seidel method [26], [27].

For the solution of the linear equations, relaxation methods can replace direct methods for the solution of (4). Let  $A$  be split into  $L + D + U$ , where  $L \in \mathbb{R}^n$  is strictly lower triangular,  $D \in \mathbb{R}^n$  is diagonal, and  $U \in \mathbb{R}^n$  is strictly upper triangular. Then the two methods mentioned earlier have the following form when applied to the solution of (4):

Gauss–Jacobi:

$$Dx^{k+1} = -(L + U)x^k + b \quad (5a)$$

or

$$x^{k+1} = -D^{-1}((L + U)x^k - b) \triangleq M_{GJ}x^k + D^{-1}b \quad (5b)$$

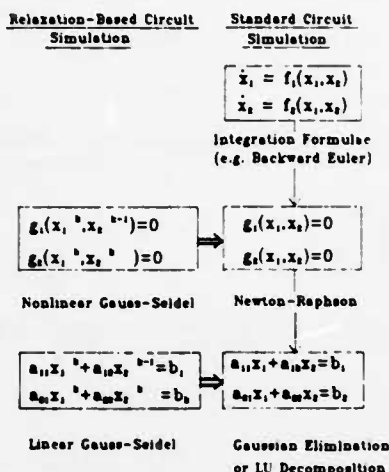


Fig. 3. Parallel between standard circuit simulation techniques and relaxation-based techniques.

*Gauss-Seidel:*

$$(L + D)x^{k+1} = -Ux^k + b \quad (6a)$$

or

$$x^{k+1} = -(L + D)^{-1}(Ux^k - b) \triangleq M_{GS}x^k + (L + D)^{-1}b \quad (6b)$$

where  $x^k$  is the value of  $x$  at the  $k$ th iteration.

Since relaxation methods are iterative methods, it is important to ask under what conditions they are guaranteed to converge to the solution of (4).

Note that the iterations are not well defined if  $D$  is singular. That is, if there is a zero on the main diagonal of  $A$ . It is well known that a necessary and sufficient condition for the iterations defined by (5b) and (6b) to converge to the solution of (4), independent of the initial guess  $x_0$ , is that the eigenvalues of  $M_{GS}$  and  $M_{GS}$  be inside the unit circle in the complex plane [26]. However, this condition is not practical from a computational point of view and other conditions, in general sufficient conditions, are used to check the convergence of these methods. In particular, it can be shown that if  $A$  is strictly diagonally dominant, then both the Gauss-Jacobi and the Gauss-Seidel iteration converge to the solution of (4). Other sufficient conditions can be found in [26], [28].

Another important convergence property of iterative methods is rate of convergence. It can be shown that if the Gauss-Jacobi and the Gauss-Seidel iteration converge, they converge at least linearly. That is, after a sufficiently large number of iterations, the error at each iteration decreases according to

$$\|x^{k+1} - \hat{x}\| \leq \epsilon \|x^k - \hat{x}\|$$

where  $\hat{x}$  is the solution of (4).

The computational cost of both of these methods is  $O(N)$ , compared with  $O(N^{1.1-1.5})$  for direct, sparse-matrix techniques. Thus relaxation methods are advantageous from a computational point of view with respect to sparse-matrix techniques only if the number of iterations needed to obtain convergence is of the order of  $N^{0.1}$ . In addition,

sparse-matrix techniques are based on Gaussian elimination or  $LU$  decomposition and, if exact arithmetic is used, they obtain the exact solution of (4) in one step. Relaxation techniques, as mentioned previously, are not guaranteed to converge. Reliability is the basic reason why sparse-matrix techniques have been used more frequently than relaxation techniques in conventional circuit simulators. If the Gauss-Seidel method is used, reordering of the equations has an effect on the number of iterations needed to obtain a solution of (4). For example, if  $A$  is upper triangular,  $N$  iterations are needed to obtain the exact solution of (4). However, if  $A$  is reordered into lower triangular form, the solution of (4) is obtained in a single iteration. If the Gauss-Jacobi iteration is used, reordering of the equations has no effect on the speed of the algorithm.

The Gauss-Seidel method can be shown to converge faster than the Gauss-Jacobi method on a class of problems [26].<sup>1</sup> For example, if  $A$  is lower triangular, Gauss-Seidel converges to the exact solution of (4) in one iteration while Gauss-Jacobi converges in  $N$  iterations. However, the fact that at each iteration each  $x_i^{k+1}$ ,  $i = 1, \dots, N$ , does not depend on any  $x_j^{k+1}$ ,  $j = 1, \dots, N$ ;  $j \neq i$  in the Gauss-Jacobi method means that the computation of all  $x_i^{k+1}$ ,  $i = 1, \dots, N$  can proceed in parallel. This method is, therefore, well suited to modern multiprocessor computers.

#### D. Nonlinear Relaxation Methods

Relaxation methods can also be used at the nonlinear-equation solution level to augment the Newton-Raphson method, and hence replace the linear-equation solution based on sparse-matrix techniques. Let  $x^k$  denote the value of  $x$  at the  $k$ th iteration. The Gauss-Jacobi and Gauss-Seidel algorithms when applied to (3) have the following form:

*Nonlinear Gauss-Jacobi Algorithm:*

$$\begin{aligned} &\text{repeat } \{ \text{forall } (j \text{ in } N) \{ \\ &\quad \text{solve } g_j(x_1^k, \dots, x_j^{k+1}, \dots, x_N^k) = 0 \text{ for } x_j^{k+1}; \} \\ &\text{until } (\|x^{k+1} - x^k\| \leq \epsilon) \end{aligned} \quad (7)$$

that is, until convergence is obtained. The *forall* ( $i$  in  $J$ ) construct specifies that the computations for all values of  $i$  in the set  $J$  may proceed concurrently, i.e., in parallel and in any order.

*Nonlinear Gauss-Seidel Algorithm:*

$$\begin{aligned} &\text{repeat } \{ \text{foreach } (j \text{ in } N) \{ \\ &\quad \text{solve } g_j(x_1^{k+1}, \dots, x_j^{k+1}, \dots, x_N^k) = 0 \text{ for } x_j^{k+1}; \} \\ &\text{until } (\|x^{k+1} - x^k\| \leq \epsilon) \end{aligned} \quad (8)$$

The *foreach* ( $i$  in  $J$ ) construct specifies that the computations for each value of  $i$  in the ordered set  $J$  must proceed sequentially and in the order specified by the set. For this

<sup>1</sup>Note that examples can be found where Gauss-Jacobi converges faster than Gauss-Seidel.

method the actual order in which the node equations are solved may be determined either statically or dynamically, as described later in Subsection III-B.

The nonlinear Gauss-Jacobi and Gauss-Seidel iterations are well defined only if each equation described in (7) and (8) has a unique solution in some domain under consideration. In the linear case, the iterations were well defined if  $D$  was nonsingular. In the nonlinear case we have a similar condition. In addition, the conditions under which these methods converge are also analogous to the ones given for the linear case.

Let  $g'(x)$  denote the Jacobian of  $g$  computed at  $x$ . Let  $g$  be continuously differentiable in an open neighborhood  $S_0$  of  $\hat{x}$  for which  $g(\hat{x}) = 0$ . Let  $g'(\hat{x})$  be split as  $L(\hat{x}) + D(\hat{x}) + U(\hat{x})$  where  $L(\hat{x})$ ,  $D(\hat{x})$ , and  $U(\hat{x})$  are, respectively, the strictly lower triangular part, the diagonal part and the strictly upper triangular part of  $g'(\hat{x})$ . Let  $M_{GJ}(\hat{x})$  and  $M_{GS}(\hat{x})$  be defined as follows:

$$M_{GJ}(\hat{x}) = -D(\hat{x})^{-1}(L(\hat{x}) + U(\hat{x})) \quad (9)$$

and

$$M_{GS}(\hat{x}) = -(D(\hat{x}) + L(\hat{x}))^{-1}U(\hat{x}). \quad (10)$$

Assume that  $D(\hat{x})$  is nonsingular and that all the eigenvalues of  $M_{GJ}(\hat{x})$  and  $M_{GS}(\hat{x})$  are inside the unit circle. Then there exists an open ball  $S \subset S_0$  such that the nonlinear Gauss-Jacobi and the Gauss-Seidel iterations are well defined and for any  $x_0 \in S$ , the sequence generated by the iterations converges to  $\hat{x}$ .

This result assumes that (7) and (8) can be solved exactly. Since these equations are nonlinear, there is no hope of computing the solutions exactly in finite time. Therefore an iterative method must be used. In general, the Newton-Raphson method is used to solve these equations. Note that for each relaxation iteration,  $N$  decoupled equations, each in one unknown, must be solved. Thus the implementation of the Newton-Raphson method is straightforward. These "composite" methods are called the Gauss-Jacobi-Newton and Gauss-Seidel-Newton methods to specify that the Newton iteration is performed inside the nonlinear Gauss-Jacobi and Gauss-Seidel iterations, respectively [27].

It is important to determine when to stop the iteration of the "inner" Newton-Raphson loop to achieve the same convergence as in the ideal case when the solutions of (7) and (8) are computed exactly. It turns out rather surprisingly that *one iteration only of the Newton method on (7a) and (8a) is sufficient to preserve the convergence properties of the nonlinear relaxation methods* [27]. In particular, the rate of convergence of the nonlinear Gauss-Seidel method is the same as the rate of convergence of the Gauss-Seidel-Newton method.<sup>2</sup>

<sup>2</sup>Note that rate of convergence is an asymptotic measure of the speed of the algorithm, i.e., of the number of iterations needed to achieve a given accuracy. Performing additional iterations of the inner Newton-Raphson loop may make the outer relaxation loop converge in fewer iterations, in some cases.

Note that the convergence result presented above is local in the sense that the iterations are guaranteed to converge only if the initial guess is sufficiently close to a solution. In this respect, the convergence properties of relaxation methods are similar to the ones of the Newton-Raphson method. However, the eigenvalue condition of relaxation methods is much stronger than the other conditions of Newton-Raphson methods. Moreover, the rate of convergence of relaxation methods is only linear while it is quadratic for Newton-Raphson methods. This explains why Newton-Raphson methods are preferred in standard circuit simulation. However, each iteration of a relaxation method involves a set of decoupled equations while Newton-Raphson methods require the solution of a set of simultaneous equations. In addition, relaxation methods are ideally suited to exploit the latency of the circuit under analysis as described in the following sections.

A comparison can be made of the use of relaxation methods at the linear and nonlinear equation level. If the relaxation methods are applied at the linear equation level and the iteration of the inner relaxation loop is carried to convergence, then the convergence of the Newton methods is not affected. However, if the inner loop is not carried to convergence, but a fixed number of iterations is allowed, then the convergence of the outer Newton loop is affected. In fact, if only one iteration of the inner relaxation loop is taken, then the convergence of the "Newton-Gauss" methods is only linear. If more iterations are taken, then the rate of convergence asymptotically improves to be quadratic [27]. The use of relaxation at the linear equation level involves the computation of the Jacobian of  $g$ , which is quite expensive as mentioned earlier. Nonlinear relaxation methods coupled with an inner Newton-Raphson loop only need the computation of the partial derivative of  $g$ , with respect to  $x_i$ , resulting in a considerable saving of computer time per iteration.

As in the linear case, the Gauss-Seidel method tends to converge faster than Gauss-Jacobi. Reordering of the equations affects the speed of the Gauss-Seidel method crucially. In this task, the *dependency matrix* of (3) plays an important role. The dependency matrix is defined to be a zero-one matrix  $P = [p_{ij}]$  such that  $p_{ij} = 1$  if  $g_i$  depends on  $x_j$ ,  $p_{ij} = 0$  otherwise. Note that  $P$  also represents the zero-nonnzero structure of the Jacobian of  $g$ .

If  $P$  is lower triangular, then only one iteration of the outer Gauss-Seidel relaxation loop is needed, provided that the inner Newton-Raphson loop is run to convergence. If  $P$  is not lower triangular, but the dependency of the  $g_i$  component of  $g$  on  $x_j$ ,  $j < i$ , is "weak," then the Gauss-Seidel method converges rather quickly. Then a key issue in applying relaxation techniques to the solution of circuit equations is the reordering of the equations so that  $P$  is almost lower triangular. This task can be performed both statically and dynamically, as described in the next section. Since MOS devices are almost unidirectional from gate to drain and gate to source due to the electrical decoupling between the gate and the source and drain of the device, and if all capacitors used in the simulation have

one node tied to a ground and the circuit does not contain any MOS transmission gates and no feedback connections, then the equations can be reordered statically to yield a lower triangular  $P$ . This property provides an intuitive explanation as to why relaxation methods are successful for the simulation of MOS digital circuits.

### E. Conclusions

To conclude this preliminary section, note that in Fig. 3 there is a "hole" in the relaxation counterpart of the flow diagram of standard circuit simulation at the differential equation level. Until recently, relaxation techniques had been used only at the linear and nonlinear equation levels. The waveform relaxation method, presented in Section V, fills that gap.

## III. TIMING SIMULATION

### A. Introduction

The first successful application of relaxation methods to electrical-circuit analysis was in timing simulation [9]–[12]. In timing simulation, only one relaxation iteration is performed per time step while one or more Newton–Raphson iterations may be performed to solve each nodal equation. Since the relaxation loop is not taken to convergence, a small time step must be used to bound local errors and the saturating properties of digital MOS circuits are exploited to bound error propagation. Timing simulators have proved successful when applied to constrained IC design methods, such as standard cell [31] or gate array, but have not been as successful in the custom-design environment. Since there is no way to guarantee accuracy for an arbitrary connection of MOSFET's unless at least two relaxation iterations are performed per time step, timing simulators have produced incorrect results in some situations. A circuit designer will use a program that gives the correct simulation result and occasionally gives no result (e.g., no convergence at a time point). A circuit designer soon loses confidence in a program that occasionally gives an incorrect answer! Many timing simulators that were developed in-house in industry are no longer in use although where they do remain in use, they continue to be very successful. When used correctly, timing simulators can provide over two orders of magnitude speed improvement over conventional circuit simulators for comparable waveform accuracy.

As described in detail later, timing simulation has problems analyzing circuits containing tight feedback loops, pass transistors or floating elements.<sup>3</sup> In particular, floating capacitors are not handled satisfactorily. Early timing simulators avoided the problem of analyzing circuits with floating capacitors by not allowing the user to include them in the circuit description. Hence, it is assumed here that the nodal capacitance matrix is diagonal, that it is nonsingular for the entire range of node voltages of interest (this implies nonzero grounded capacitances), and that the

circuit equations are written as

$$\dot{v} = -C(v, u)^{-1}f(v, u) = -F(v, u). \quad (11)$$

Algorithms used for timing analysis often discretize the derivative operator by Backward Euler [9], [10], [30] or the Trapezoidal Rule [29]. For the sake of simplicity, in the following description the Backward Euler formula will be used:

$$\dot{v}_{k+1} = (v_{k+1} - v_k)/h,$$

where the time step  $h = t_{k+1} - t_k$  and  $v_{k+1}$  and  $v_k$  are the computed values of the node voltages at time  $t_{k+1}$  and  $t_k$ , respectively. The solution of the resulting nonlinear system of equations

$$v_{k+1} - v_k + hF(v_{k+1}, u(t_{k+1})) = 0 \quad (12)$$

is then approximated by one sweep of a relaxation technique.

Program MOTIS [9] used a modified Gauss–Jacobi technique which yields the following set of decoupled equations:

$$v_{k+1}^n = v_k^n - hF_n(v_k^1, \dots, v_k^{n-1}, v_{k+1}^n, v_k^{n+1}, \dots, v_k^N, u_n(t_{k+1})), \quad n = 1, \dots, N \quad (13)$$

The solution of the decoupled nonlinear equations of (6) is then approximated by taking a single step of a *regula falsi* iteration [32].

The MOTIS-C [10] and SPLICE1 [29] programs use a modified Gauss–Seidel technique. In SPLICE this technique yields

$$v_{k+1}^n = v_k^n - hF(\bar{v}_{k+1,n}, u(t_{k+1})), \quad n = 1, 2, \dots, N \quad (14)$$

where

$$\bar{v}_{k+1,n} = [v_{k+1}^1, \dots, v_{k+1}^n, v_k^{n+1}, \dots, v_k^N]^T. \quad (15)$$

The solution of (14) is then approximated by using one or more steps of the Newton–Raphson algorithm. The program can cut the time step locally at a node and use a number of small time steps to achieve a satisfactory solution before the next equation in the relaxation solution is to be processed.

### B. Network Ordering

Unless some form of connection graph is used to establish a precedence order for signal flow, the new node voltages will be computed in an arbitrary order. As pointed out in Section II, in a Gauss–Jacobi-based simulator, where only node voltages at  $t_k$  are used to evaluate the node voltage at  $t_{k+1}$ , the order of processing elements will not affect the results of the analysis. However, substantial timing errors may occur. For example, consider the inverter chain of Fig. 4. If the input to inverter  $I_1$  changes at time  $t_k$ , that change cannot appear at node (1) before time  $t_{k+1}$ . For a chain of  $N$  inverters, the change will not appear at output  $I_N$  before  $N$  time steps have elapsed. If the time step is very small with respect to the response time of any one inverter, this error may not be significant.

<sup>3</sup>A floating element is a two-terminal capacitor or resistor whose terminals are not ground or power supply.



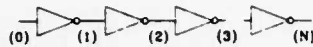


Fig. 4. Inverter chain example.

In a Gauss-Seidel-based simulator, where node voltages already computed at  $t_{k+1}$  are made available for the evaluation of other node voltages at  $t_{k+1}$ , the order of processing elements can affect the simulator performance substantially. In the previous example, if the inverters were processed in the order 1, 2, 3, ...,  $N$  then  $v_{k+1}^1$  can be determined from  $v_{k+1}^0$ ,  $v_{k+1}^2$  from  $v_{k+1}^1$ , and so on. The result will be zero accumulated timing error. Should the nodes happen to be processed in the reverse order,  $N, N-1, \dots, 1$ , then a timing error of  $N$  time steps will occur, the same error as in the Gauss-Jacobi-Newton iteration.

If it were possible to order the processing of nodes in the Gauss-Seidel-Newton iteration so as to follow the flow of the signal through the circuit, the timing error would be kept small. A signal flow graph would provide this information. An example of a circuit fragment and associated signal flow graph, illustrating the fanins and fanouts of the nodes, is shown in Fig. 5. One way to generate this graph is to consider the dependency matrix introduced in Section II. This zero-one matrix can be considered as the adjacency matrix of a directed graph  $G = G(X, E)$ , where  $X$  is the set of vertices and  $E$  is the set of directed edges of the graph. An edge connects nodes  $x_i$  to node  $x_j$  if  $p_{ij} = 1$ . By the definition of dependency matrix, given a circuit and its node equations written as in (11), this graph indicates that if an edge connects  $x_i$  to  $x_j$ , then the voltage of node  $i$ ,  $v_i$ , can affect the value of the voltage of node  $j$ ,  $v_j$  via the device equation. Thus the set of vertices that are connected by an edge going in  $x_i$  identifies all the nodes in the circuit that affect the value of the voltage of node  $i$ . These are called *fanin nodes* of node  $i$ . Similarly, the set of nodes that are connected to node  $x_i$  by an edge going out of  $x_i$  identifies all the nodes in the circuit whose voltage is affected by the voltage of node  $i$ . These are called *fanout nodes* of node  $i$ . Note that a node can be both a fanin and a fanout node of node  $i$ .

Fanin and fanout elements can also be defined. The fanin elements of node  $i$  are defined as those which play some part in determining the voltage at node  $i$ , i.e., those elements that cause some entries of row  $i$  in the dependency matrix to be one. For example, any MOS transistor, modeled by the simple Shichman-Hodges [33] equations shown in Fig. 6, whose drain or source is connected to a node would be classified as a fanin element at that node since its drain or source current may affect the node voltage.

A fanout element of node  $i$  is one whose operating conditions are directly influenced by the voltage at node  $i$ , i.e., those elements that cause some entries of column  $i$  in the dependency matrix to be one. For MOS transistors, connection to any of the three independent ports (drain, gate, or source) would cause that MOS transistor to be included in the fanout-element set at the node. It is therefore possible for an element to appear as both a fanin and a fanout at the node.

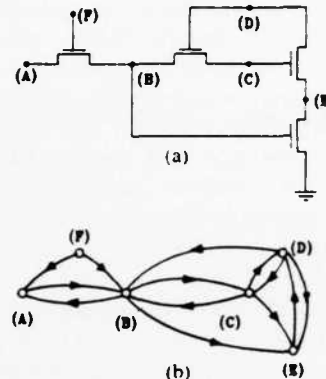
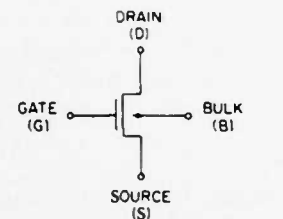


Fig. 5. (a) Circuit fragment. (b) Associated signal flow graph.



$$\begin{aligned}
 V_t &= V_{TO} + \gamma(\sqrt{V_{bs} + \phi} - \sqrt{\phi}) \\
 V_{gs} &= V_{gs} - V_t \\
 V_{ds} &= \text{MAX}(V_{ds}, V_{gs}) \\
 I_{ds} &= K(V_{gs} - V_{tse}/2)V_{ds}(1 + \lambda V_{ds})
 \end{aligned}$$

(b)

Fig. 6. (a) n-channel MOS transistor. (b) Simple Shichman-Hodges n-channel MOS model equations.

SPLICE1 builds the signal flow graph by constructing two tables for each node as the circuit is read. First, all circuit elements are classified as fanin and/or fanout elements of the nodes to which they are connected. The two tables constructed for each node contain the names of the fanin and fanout elements at the node. These tables are generated as the elements are read into memory. If this graph is acyclic, then it can be *levelized* [12, p. 427], where a level number corresponding to the longest path (most branches traversed) from any independent source to a node is assigned to each node. If the graph does contain cycles, special steps must be taken to break these feedback loops. Then if the nodes are processed in the order of level numbers, it is clear that an optimal static ordering for Gauss-Seidel processing will be achieved. An ordering can also be achieved by finding the strongly connected components of the graph [34]. The levelized graph provides a *static* ordering of the network.

Whenever the voltage at a node changes, it is possible to *schedule* all of its fanouts to be processed. In this way, the effect of a change at the input to a circuit may be *traced* as it propagates to other circuit nodes via the fanout tables, and thus via the circuit elements which are connected to them. Since the only nodes processed are those which are affected directly by the change, this technique is *selective* and hence its name: *selective trace*. If a selective-trace algorithm is used with the fanin and fanout tables, the

order in which the node voltages are updated becomes a function of the signals flowing in the network, and is therefore a dynamic ordering. This approach is often used in modern logic simulators and is also the ordering technique used in the SPLICE program.

Even with selective trace some timing errors can occur. For example, wherever feedback paths exist, one time step of error may be introduced. Consider the circuit fragment and its associated signal flow graph shown in Fig. 5. Assume  $v_1$  and  $v_2$  are such that both  $M_1$  and  $M_2$  are conducting. If a large input appears at node (1) at time  $t_{k+1}$ , it will be traced through nodes (1), (2), (3), and (4), respectively. Now, however, the change in voltage at node (4) caused node (1) to be marked to be processed again at this time. Since only one sweep of Gauss-Seidel is being used, the solution of node (1) a second time would be illegal. Rather, the node (1) is scheduled to be processed one time step in the future, and thus it is possible that one time step of timing error has been introduced.

### C. Exploiting Latency

As mentioned earlier, large digital circuits are often relatively inactive. A number of schemes can be used to avoid the unnecessary computation involved in the reevaluation of the voltage at nodes which are inactive or latent.

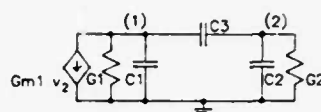
A scheme used in many electrical simulators is the "bypass" scheme, described in Section II for conventional circuit simulators. This scheme has also been employed in a number of timing simulators. However, when the majority of the nodes in the circuit are latent, the task of simply checking each node to determine if it can be bypassed can dominate the total run time.

The use of the selective-trace technique for dynamic ordering can provide a major time saving here. By constructing a list of all nodes which are active at a time point and excluding those which are not, selective trace allows circuit latency to be exploited without the need to check each node for activity. The elimination of this checking process, used in both the bypass approach and the static leveling scheme described previously, can save a significant amount of computer time for large circuits at the cost of some extra storage for the fanin and fanout tables at each node.

In an efficient implementation of the selective-trace technique, the fanin and fanout tables do not contain the "names" of fanin and fanout elements, respectively, but rather a *pointer* to the actual location in memory where the data for each element is stored.

### D. Numerical Properties of Timing Algorithms

A major drawback with the use of timing analysis is that tightly coupled feedback loops, or bidirectional circuit elements, can cause severe inaccuracies and even instability during the analysis. For example, if the Gauss-Seidel "one-sweep" timing-analysis method is applied to the circuit of Fig. 7 limiting the time-step to 0.1 s, the waveforms of Fig. 8(a) are obtained. However, if the time step is set to 0.8 s, then the computed solution blows up as shown in



$$G1=G2=1\text{mho}; C1=C2=1\text{F}; Gm1=15\text{mho}$$

Fig. 7. Schematic diagram of the example circuit.

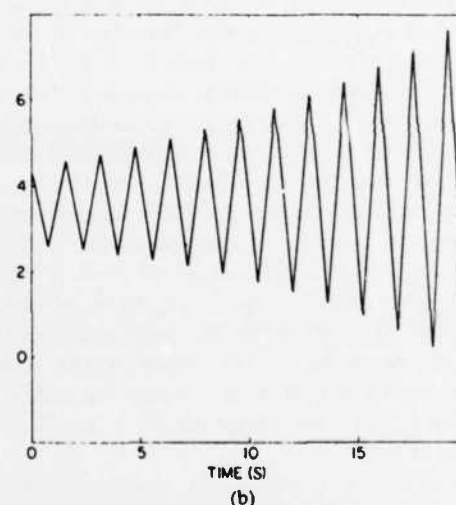
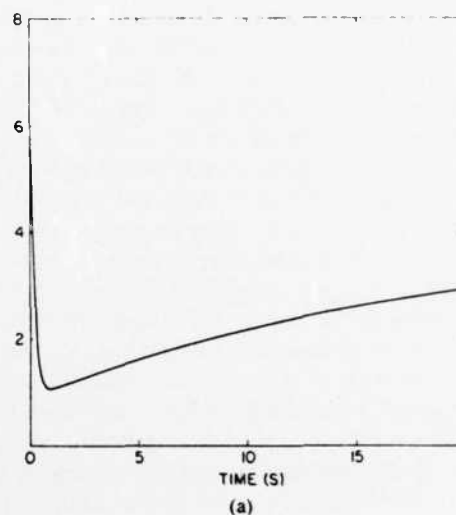


Fig. 8. (a) Accurate waveform of voltage  $v_1$  computed with a 0.1-s time-step. (b) Waveform of voltage  $v_1$  computed with 0.8-s time-step.

Fig. 8(b). This demonstrates that timing algorithms do not inherit the numerical properties of the discretization formulae used to approximate the time derivative. In fact, Backward Euler is well known to be  $A$  stable, i.e., the computed solution of the circuit differential equations should not "blow up" independent of the choice of time step as long as the simulated circuit is stable.

The reason why this idiosyncrasy is observed is that timing algorithms do not solve (5) since only one sweep of the relaxation iteration is taken. Therefore the stability and accuracy properties of the integration method used to discretize the derivative operator no longer hold. As a matter of fact, the combination of the discretization formula, the various relaxation steps, and the Newton-Raphson method form a set of new integration algorithms.

These integration methods use an implicit formula to discretize the differential equations, but they do not solve the nonlinear equation obtained. Thus they are somewhat in between explicit and implicit methods.

In the following description, the "time-advancement" algorithms which use the Gauss-Jacobi and the Gauss-Seidel relaxation step will be referred to as *Gauss-Jacobi* and *Gauss-Seidel integration algorithms*, respectively. This perspective allows the understanding of the numerical behavior of timing algorithms and the development of better techniques for timing simulation.

An analysis of numerical properties of the Gauss-Jacobi and Gauss-Seidel integration algorithms when applied to MOS circuits has been carried out in [35], [36]. Only the most important results are outlined here. First consider the case where no floating capacitors are present in the circuit to be analyzed.

The numerical properties of an integration method, such as stability, are studied on test problems [37], [38], which are simple enough to allow a theoretical analysis but still sufficiently general that some insight can be obtained about how the method will behave in general. For the widely used linear multistep methods, the test problem consists of a linear time-invariant asymptotically stable autonomous differential equation. Unfortunately this simple test problem cannot be used to evaluate relaxation-based time-advancement techniques. In fact, each variable of the system of differential equations is treated differently according to the ordering in which equations are processed. Hence a more complex test problem is needed. The test problem chosen here is a linear time-invariant asymptotically stable system of autonomous differential equations, i.e.,

$$\begin{aligned}\dot{x} &= Ax \\ x(0) &= X\end{aligned}\quad (16)$$

where  $A \in R^{n \times n}$  and the set of eigenvalues (spectrum) of  $A$ ,  $\sigma(A)$ , is in the open left-half complex plane, i.e.,  $\sigma(A) \in C_0$ .

In circuit theoretic terms, linear circuits whose natural frequencies are in the open left-half plane and which satisfy the assumptions described in Section II are considered as test circuits. Let  $A = L + D + U$ , where  $L$  is strictly lower triangular,  $D$  is diagonal, and  $U$  is strictly upper triangular. The time-advancement methods presented in Section III applied to the test system of (16) yield the following recursive relations:

*Gauss-Jacobi integration algorithm:*

$$[I - hD]x_{k+1} = [I + h(L + U)]x_k \quad (17)$$

$$x_{k+1} = M_{GJ}(h)x_k \quad (18)$$

where  $I$  is the identity matrix and

$$M_{GJ}(h) = [I - hD]^{-1}[I + h(L + U)]. \quad (19)$$

*Gauss-Seidel integration algorithm:*

$$[I - h(D + L)]x_{k+1} = [I + hU]x_k \quad (20)$$

$$x_{k+1} = M_{GS}(h)x_k \quad (21)$$

where

$$M_{GS}(h) = [I - h(D + L)]^{-1}[I + hU]. \quad (22)$$

The matrices  $M_{GJ}(h)$  and  $M_{GS}(h)$  are called the companion matrices of the methods. If the generic companion matrix of a method is denoted  $M(h)$ , then

$$x_k = [M(h)]^k x_0. \quad (23)$$

The numerical properties of the integration algorithms described by (23) are now described following the outline of one-step integration methods applied to ordinary differential equations [37].

The first numerical property of the integration methods to investigate is accuracy. This property relates the error introduced by the discretization process and the time step.

*Definition III-D-1:* Let  $x(t_k)$  be the exact value of the solution of the test problem at time  $t_k$ . Let  $x_k$  be the computed solution at time  $t_k$  assuming  $x_{k-1} = x(t_{k-1})$ , i.e., that no error has been made in computing the value of  $x$  at the previous time point. If  $h = t_k - t_{k-1}$ , the local truncation error is defined to be

$$\epsilon = \|x(t_k) - x_k\|. \quad (24)$$

If  $\epsilon = O(h^{r+1})$ ,  $r$  is said to be the order of the integration method [37]. ■

It has been observed experimentally that if the time step is decreased, the accuracy of the solution computed by timing algorithms improves in almost the same way as Backward Euler. In fact, the Gauss-Jacobi and the Gauss-Seidel integration algorithms have the same accuracy as the Backward Euler integration method.

*Theorem III-D-2:* Gauss-Jacobi and Gauss-Seidel integration algorithms are first order integration algorithms. ■

In circuit analysis, another important criterion for evaluating the accuracy of an integration method, can be defined as *waveform accuracy*. In general, the computed solution of a system of differential equations is the superposition of a principal solution and associated parasitic solutions. Parasitic solutions are generated by the numerical approximations of the integration methods. In particular, an  $n$ th order integration algorithm yields  $n-1$  parasitic solutions when applied to the test problem. For the algorithms under consideration in this paper, the displacement techniques introduce additional spurious components called *numerical solution components*.

If the original system to be analyzed does not contain an oscillatory component, the presence of such a component in the computed solution can be misleading in the evaluation of the performances of the system. As was shown in the previous subsection, the Gauss-Seidel integration algorithm introduces spurious oscillations in the computed solution of the equations describing a circuit where the exact solution does not have any oscillations. It is necessary to introduce methods that allow the evaluation of the "waveform accuracy" of the integration methods. To this end, a subclass of the test problem is now introduced, characterized by  $\sigma(A) \in R_0$ , i.e., the set of test problems



which does not have an oscillatory component in the solution, and bounds on the oscillatory components of the computed solutions must be established.

Theorem III-D-2 provides a bound on the oscillatory components of all the methods. In particular it is clear that, by choosing an appropriately small step size  $h$ , the numerical solution oscillatory components can be made negligible with respect to the principal solution.

Another fundamental property of integration methods is stability. Stability can also be defined in terms of the test problem.

**Definition III-D-3 (Stability):** An integration algorithm is stable if  $\exists \delta > 0, \exists N > 0$  such that  $\forall x_0 \in R^n, \exists \bar{k} > 0$  and

$$\|x_k\| < N, \quad \forall k \geq \bar{k}, \quad \forall h \in [0, \delta] \quad (25)$$

where  $x_k$  is the sequence generated by the algorithm applied to the test problem according to (23). ■

It is obvious that a numerical method that is not stable is of no practical use. It happens that both relaxation-based integration algorithms are stable as stated in the following theorem which is proven in [35].

**Theorem III-D-4:** Gauss-Jacobi and Gauss-Seidel integration algorithms are stable. ■

The accuracy and the stability of the integration algorithms explain the success of timing simulators, but they also point out what the problems are with their use. In particular, note that  $\delta$  of Definition III-D-3 can be quite small, i.e., that the time step may have to be reduced not for accuracy reasons, but to make sure that the computed solution does not blow up. Moreover, it is difficult to identify oscillations in the computed solutions as spurious.

To cope at least in part with these problems, another displacement technique for the solution of (1) has been proposed for a simple circuit in [42]. This algorithm is a symmetric-displacement method reminiscent of the alternating-direction implicit method [32] and is based on a class of methods proposed by Kahan [39]. The basic idea here is to "symmetrize" the Gauss-Seidel scheme with a method that takes two half-steps of size  $\frac{h}{2}$  each: one half-step is taken in the usual "forward" (i.e., lower triangular) direction, the second half-step is taken in the backward (i.e., upper triangular) direction.

This method is introduced with the help of the linear system described by (16). The first half-step corresponds to the Gauss-Seidel method. Let  $A = L + D + U$ , then

$$\left(I + \frac{h}{2} \left(\frac{1}{2}D + L\right)\right) x_{k+1/2} = \left(I - \frac{h}{2} \left(\frac{1}{2}D + U\right)\right) x_k. \quad (26)$$

Note that there is a difference between (20) and (26) since  $D$  has been split into two parts here. This splitting of  $D$  is necessary to "symmetrize" the method. The backward half-step is then

$$\left(I + \frac{h}{2} \left(\frac{1}{2}D + U\right)\right) x_{k+1} = \left(I - \frac{h}{2} \left(\frac{1}{2}D + L\right)\right) x_{k+1/2}. \quad (27)$$

Consider the simple example of Fig. 9. The first half-step

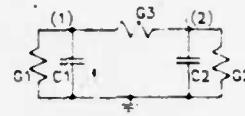


Fig. 9. Simple circuit to illustrate the modified symmetric Gauss-Seidel method

yields

$$\begin{aligned} \left(1 + \frac{h}{4} \frac{G_1 + G_3}{C_1}\right) x_{k+1/2}^1 &= \left(1 - \frac{h}{4} \frac{G_1 + G_3}{C_1}\right) x_k^1 + \frac{h}{2} \frac{G_3}{C_1} x_k^2 \\ \left(1 + \frac{h}{4} \frac{G_2 + G_3}{C_2}\right) x_{k+1/2}^2 &= \frac{h}{2} \frac{G_3}{C_2} x_{k+1/2}^1 \\ &\quad + \left(1 - \frac{h}{4} \frac{G_2 + G_3}{C_2}\right) x_k^2. \end{aligned}$$

The backward step involves the solution of

$$\begin{aligned} \left(1 + \frac{h}{4} \frac{G_2 + G_3}{C_2}\right) x_{k+1}^2 &= \frac{h}{2} \frac{G_3}{C_2} x_{k+1/2}^1 \\ &\quad + \left(1 - \frac{h}{4} \frac{G_2 + G_3}{C_2}\right) x_{k+1/2}^2 \\ \left(1 + \frac{h}{4} \frac{G_1 + G_3}{C_1}\right) x_{k+1}^1 &= \frac{h}{2} \frac{G_3}{C_1} x_{k+1}^2 \\ &\quad + \left(1 - \frac{h}{4} \frac{G_1 + G_3}{C_1}\right) x_{k+1/2}^1. \end{aligned}$$

If these formulas are generalized for the nonlinear case, and with

$$\bar{v}_{i,i} = \begin{bmatrix} v_i^1, \dots, v_i^i, v_{i-1/2}^{i+1}, \dots, v_{i-1/2}^n \end{bmatrix}^T \quad \text{if } 2i \text{ is odd} \quad (28)$$

$$\bar{v}_{i,i} = \begin{bmatrix} v_{i-1/2}^1, \dots, v_{i-1/2}^i, v_i^1, \dots, v_i^n \end{bmatrix}^T \quad \text{if } 2i \text{ is even.} \quad (29)$$

the forward step yields

$$\begin{aligned} v_{k+1/2}^i - v_k^i + \frac{h}{4} F_i(\bar{v}_{k+1/2,i}, u(t_{k+1/2})) \\ + \frac{h}{4} F_i(\bar{v}_{k+1/2,i-1}, u(t_{k+1/2})) = 0, \\ i = 1, 2, \dots, N \quad (30) \end{aligned}$$

and the backward step yields

$$\begin{aligned} v_{k+1}^i - v_k^i + \frac{h}{4} F_i(\bar{v}_{k+1,i}, u(t_{k+1})) \\ + \frac{h}{4} F_i(\bar{v}_{k+1,i+1}, u(t_{k+1})) = 0, \\ i = N, N-1, \dots, 1. \quad (31) \end{aligned}$$

The solution of the decoupled equations is then approximated by taking one step of the Newton-Raphson algorithm.

This method can be proven to be more accurate and more stable than the previous one. The additional work required to perform the intermediate step is compensated by the additional accuracy as specified by the following theorem.

**Theorem III-D-5:** The modified symmetric Gauss-Seidel algorithm is a second-order integration algorithm. ■

In addition, the "waveform accuracy" of this method is better than that of the other integration algorithms. If the class of the test problems is restricted to the subclass characterized by a symmetric  $A$  matrix, then a strong result for the modified symmetric Gauss-Seidel integration method can be obtained. In circuit theoretic terms, this analysis applies to linear circuits whose node equations yield a symmetric nodal admittance matrix when only the resistive part of the circuit is considered. Moreover it is required that this matrix remain symmetric when premultiplied by  $C^{-1}$ , the diagonal matrix of the grounded capacitors. A sufficient condition for this to occur is that the circuit consists of two terminal linear resistors and capacitors and that the grounded capacitors be of equal value. The case where the capacitors are not of equal value can also be included in this class provided that a scaling of the rows of the matrix is performed.

**Theorem III-D-6:** If  $A$  is a real, symmetric matrix, the spectrum of the companion matrix of the modified symmetric Gauss-Seidel integration method is real, i.e., no oscillatory parasitic components are present in the computed solution. ■

This theorem guarantees that the time step does not have to be limited to eliminate spurious oscillations at least for reciprocal circuits. Numerical results obtained with an experimental timing simulator show that spurious oscillations do not appear in the solution of nonlinear nonreciprocal circuits as well [36], [40].

For computational efficiency, it is desirable that the step size be limited only by accuracy considerations as in the case of the implicit backward differentiation formulas [37]. In the case of classical multistep methods, the concept of  $A$ -stability [38] and stiff-stability [37] have been introduced to test the "unconditional" stability of multistep methods. For the "time-advancement" techniques presented in this paper, it makes sense to define a similar concept. Unfortunately, general results of "unconditional" stability are not available for the test problem defined previously, but only for a subclass; once more the subclass characterized by a symmetric  $A$  matrix.

**Definition III-D-7 ( $\tilde{A}$  stability):** An integration method is  $\tilde{A}$  stable if  $\exists N > 0$  such that  $\forall x_0 \in R^n, \exists \bar{k}$

$$\|x_k\| < N, \quad \forall k \geq \bar{k}, \quad \forall h \in [0, \infty) \quad (32)$$

where  $\{x_k\}$  is the sequence generated by the method applied to the test problem of (11) with  $A$  symmetric. ■

**Theorem III-D-8:** The modified symmetric Gauss-Seidel method is  $\tilde{A}$  stable. ■

Note that no  $\tilde{A}$  stability result for the Gauss-Jacobi and the Gauss-Seidel integration methods has been proven. In our practical experiments, we have seen that when applied to real circuit problems, the modified symmetric Gauss-Seidel method is indeed "more stable" than the other two methods.

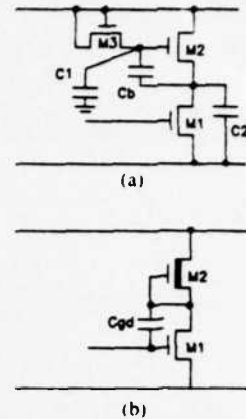


Fig. 10. (a) Bootstrap inverter circuit and (b) effect of  $C_{gd}$  feed-through.

### E. Floating Capacitors

As mentioned in the introduction to this section, a circuit element that has limited the application of timing analysis is the floating capacitor [41], [42].

The floating capacitor is often an important element in the design of integrated circuits. In Fig. 10(a) the value of the bootstrap capacitor  $C_b$  is generally large compared with the values of the associated parasitic grounded capacitors  $C_1$  and  $C_2$ . The value of the intrinsic gate-drain feedthrough capacitance  $C_{gd}$  in Fig. 10(b) is often small compared with other circuit parasitics at the gate and drain nodes; however, the effect of  $C_{gd}$  on circuit performance can be significant due to the large voltage gain of the stage.

When floating capacitors are present in the circuit to be analyzed, the timing simulation algorithms presented in the previous sections take a different form. For the sake of simplicity, consider a linear time-invariant circuit described by

$$C\dot{v} = -Gv, \quad v(0) = V \quad (33)$$

where  $C$  is the node capacitance matrix and  $G$  is the node conductance matrix. If  $C$  is inverted, the methods described previously apply. However, inverting  $C$  is expensive and most of the advantages of timing simulation algorithms would be lost. Thus if the Backward Euler formula is used to discretize the circuit equations at time  $t_{k+1}$ ,  $v_{k+1}$  is given by

$$C(v_{k+1} - v_k) = -hGv_{k+1} \quad (34)$$

or, rearranging (34),  $v_{k+1}$  is given by

$$(C + hG)v_{k+1} = Cv_k. \quad (35)$$

Let  $C$  be split as  $C_d + C_l + C_u$  and  $G$  as  $G_d + G_l + G_u$ , where  $C_d$  and  $G_d$  are diagonal matrices,  $C_l$  and  $G_l$  are strictly lower triangular matrices, and  $C_u$  and  $G_u$  are strictly upper triangular matrices. Then, the time-advancement Gauss-Jacobi algorithm for circuits described by (33) becomes

$$\begin{aligned} (C_d + hG_d)v_{k+1} &= (C - C_l - C_u - hG_l - hG_u)v_k \\ &= (C_d - h(G_l + G_u))v_k \end{aligned} \quad (36)$$

and the Gauss-Seidel time-advancement algorithm is

$$\begin{aligned} (C_d + C_l + h(G_d + G_l))v_{k+1} &= (C - C_u - hG_u)v_k \\ &= (C_l - hG_u)v_k \end{aligned} \quad (37)$$

and finally the modified symmetric Gauss-Seidel algorithm is

$$\begin{aligned} \left(\frac{1}{2}C_d + C_l + \frac{h}{2}\left(\frac{1}{2}G_d + G_l\right)\right)v_{k+1/2} \\ = \left(\frac{1}{2}C_d + C_l - \frac{h}{2}\left(\frac{1}{2}G_d + G_u\right)\right)v_k \end{aligned} \quad (38a)$$

$$\begin{aligned} \left(\frac{1}{2}C_d + C_u + \frac{h}{2}\left(\frac{1}{2}G_d + G_u\right)\right)v_{k+1} \\ = \left(\frac{1}{2}C_d + C_u - \frac{h}{2}\left(\frac{1}{2}G_d + G_l\right)\right)v_k \end{aligned} \quad (38b)$$

When these algorithms are applied to circuits where  $C$  is not diagonal, serious stability and accuracy problems may arise. In the example of Fig. 7, the Gauss-Seidel integration algorithm with a time step equal to 0.6 s computes the solution shown in Fig. 11, with oscillations that are not present in the accurate solution of the circuit equations shown in Fig. 8(a). In addition, these methods are not even consistent. That is, when the time step  $h$  is reduced to 0, the sequence of voltages computed according to (36) or to (37) does not converge to the solution of (33). In fact, since (36) is the same as the equation obtained by applying the Gauss-Jacobi algorithm to a circuit where only grounded capacitors are present, the effect of the floating capacitors is completely neglected (note that in (36),  $C_l$  and  $C_u$  do not appear). The Gauss-Seidel algorithm neglects  $C_u$  only, and hence is more accurate than the Gauss-Jacobi algorithm.

The modified symmetric Gauss-Seidel integration algorithm presented in the previous subsection has been proven to be accurate, stable, and even  $\bar{A}$  stable, but only for the particular classes of circuits characterized by  $G$  and  $C$  matrices with appropriate mathematical properties [36]. However, the modified symmetric Gauss-Seidel algorithm is not consistent in the general case either. On the other hand, since  $C_u$  is neglected in the first half-step while  $C_l$  is neglected in the second half-step, the effect of the floating capacitors on the solution of (33) is modeled more precisely than in the other methods, and better accuracy is obtained as a consequence [36].

Early timing simulators avoided the problem of analyzing floating capacitors by not allowing the user to include them in the circuit description. The effect of a floating capacitor may then be approximated by altering the values of the grounded capacitors at appropriate nodes in the circuit. If the operation of a circuit depends on a floating capacitor, a functional macromodel may be used (e.g., [9]–[11]) where the effect of the floating element is hidden from the relaxation iteration by special processing of the circuit fragment in which it is embedded, perhaps involving local matrix solution [29].

Another approach called the *Implicit-Implicit-Explicit* (IIE) method has been proposed [41] for circuits with

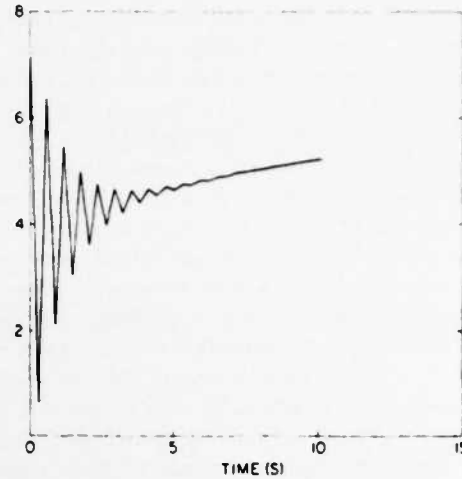


Fig. 11. Waveform of node voltage  $v_1$  computed with 0.6-s time-step.

floating capacitors. This method can be generalized and explained using (33). Let (33) be rearranged as

$$(C_d + C_l)\dot{v}' + C_u\dot{v}'' = -Gv, \quad v(0) = V \quad (39)$$

where  $\dot{v}' = \dot{v}'' = \dot{v}$ . The Backward Euler method is used to approximate  $\dot{v}'$  and Forward Euler is used to approximate  $\dot{v}''$ . Then (39) becomes

$$(C_d + C_l)(v_{k+1} - v_k) + C_u(v_k - v_{k-1}) = -hGv_{k+1}. \quad (40)$$

Applying the Gauss-Seidel integration scheme to (40) and rearranging terms, the IIE equations for the linear circuit described by (33) become

$$(C_d + C_l + h(G_d + G_l))v_{k+1} = -C_u(v_k - v_{k-1}) - hG_u v_k. \quad (41)$$

Note that the effects of both the lower triangular part and the upper triangular part of the capacitance matrix are now taken into account. However, to date the numerical properties of this scheme have only been published for a simple two-node linear test circuit [41], [42]. The method has been difficult to characterize rigorously since it involves information from two previous time points ( $t_k$  and  $t_{k-1}$ ). Recently the stability, consistency, and order of accuracy of this approach have been determined for the general case [43] and the method looks very promising. The IIE method is used in the SPLICE1 program for the analysis of floating capacitors [12].

## F. Conclusions

Timing simulation algorithms are fast and rather accurate for the electrical simulation of MOS circuits with no tight feedback loops. However, several stability and accuracy problems hamper the use of timing simulation as a standard simulation tool. In particular, major drawbacks of timing simulation algorithms are

- 1) The selection of an appropriate step size is difficult. If a fixed step size is used, heuristics must be introduced to

estimate the time constants of the circuit [10]. If a variable step size is used, the local truncation error must be estimated. In fact, the other technique used to control step size in standard circuit simulators, the so-called iteration count method [1], cannot be applied here since the relaxation techniques are not carried to convergence. Unfortunately, the local truncation error cannot be estimated accurately since the error in the voltage computed at time  $t_k$  by the timing simulation algorithms is the sum of the truncation error due to the integration method and of the error due to the inaccurate solution of the discretized nonlinear equations. These two errors can be of the same order and sometimes the latter component can even be larger than the truncation error.

2) The step size may be limited by stability considerations since timing simulation algorithms are *A* stable only in particular cases. This may force the use of small step sizes even though large step size may be possible from accuracy point of view.

3) With the exception of the IIE method, timing simulation algorithms are not even consistent when applied to the analysis of circuits containing floating capacitors. This means that their accuracy cannot be improved over a certain limit by further step-size reductions.

All of these problems stem from the fact that the relaxation methods are not carried to convergence. The fear that carrying the relaxation iteration to convergence would reduce the speed advantages of timing simulation prevented the adoption of the obvious remedy to this situation for a number of years. In the following sections, techniques and simulators based on convergent relaxation methods are introduced and shown to be highly competitive with standard circuit simulators for accuracy and with timing simulators for speed.

#### IV. ITERATED TIMING ANALYSIS

##### A. Introduction

Iterated Timing Analysis (ITA) [19] is a new form of electrical analysis which can be derived from timing analysis. This form of relaxation-based electrical analysis has shown promising results over a wide class of circuits, from large digital circuits to complex analog designs. The technique is accurate, fast for large digital circuits, and amenable to implementation on advanced computer architectures, such as vector and array processors [44]–[47] as well as data-flow machines [48], [49].

The starting point for a description of ITA is the circuit equation formulation of (2). The differential equations are converted to a set of nonlinear, algebraic difference equations (3) using a stiffly stable integration formula, and an iterative relaxation method (Gauss–Jacobi or Gauss–Seidel) is then used to solve them. However, unlike timing analysis where a single relaxation iteration is used per time point, in the ITA approach the relaxation process is continued to convergence at a time point.

Only one Newton–Raphson iteration is used to approximate the solution of each nodal equation per relaxation iteration and event-driven selective trace techniques

may still be used to exploit latency, as for timing simulation. Thus the mathematical framework of ITA is the nonlinear Gauss–Seidel (Gauss–Jacobi)–Newton method presented in Section II.

##### B. Numerical Properties of ITA

Since in ITA the nonlinear circuit equations are solved by an iterative method until satisfactory convergence is achieved, the numerical properties of the integration methods used to discretize the circuit equations are retained. Thus the stability and the accuracy problems typical of the timing simulation algorithms presented in Section III are not an issue. However, the basic question is whether the relaxation iteration will converge at each time point when solving the discretized circuit equations.

The conditions under which the relaxation iteration is guaranteed to converge were presented in Section II. Note that these conditions require the diagonal dominance of the Jacobian of the discretized nonlinear equations. Returning once again to (2), the circuit equations may be formulated as

$$C(v, u)\dot{v} + f(v, u) = 0, \quad v(0) = V \quad (42)$$

where  $C: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^{n \times n}$  is a symmetric diagonally dominant matrix-valued function in which  $-C_{ij}(v, u)$ ;  $i \neq j$  is the total floating capacitance between nodes  $i$  and  $j$ ,  $C_{ii}(v, u)$  is the sum of the capacitances of all capacitors connected to node  $i$ , and  $f: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^n$  is a continuous function, each component of which represents the net current charging the capacitor at a node due to other conductive elements. If the capacitance matrix  $C(v, u)$  is assumed to be symmetric and positive definite (and hence strictly diagonally dominant), as is the case if all the capacitors in the circuit are two-terminal elements and are positive for all values of  $v$ , it is intuitive to see that the Jacobian matrix of the discretized nonlinear circuit equations is diagonally dominant provided that the time step is small enough. In fact, the time step is acting as a scaling parameter that increases the role of the capacitance matrix in the Jacobian matrix when it is decreased. More formally, the convergence properties of ITA can be proven rather easily for circuit equations of the form of (42) where  $C(v, u)$  is a matrix of real numbers, i.e., the capacitors present in the circuit are all linear. Then the discretized equations become

$$C(v_{k+1} - v_k) - h_{k+1}f(v_{k+1}, u_{k+1}) = 0 \quad (43)$$

where  $h_{k+1}$  is the time step selected at time  $t_k$ . The following strong Theorem has been proven in [50].

**Theorem IV-B-1:** There exists a time step  $h$  strictly positive such that for all  $h_{k+1} \leq h$  the nonlinear Gauss–Jacobi and the nonlinear Gauss–Seidel iteration applied to (43) converge to the solution of the discretized circuit equations independent of the initial guess. ■

##### C. Implementation of ITA

In Theorem IV-B-1, the value of  $h$  can be quite small if the Jacobian of  $f$  is not diagonally dominant at the time point of interest and if the  $C$  matrix has large off-diagonal



elements, i.e., when large floating capacitors and/or tight feedback loops are present in the circuit. Hence, such an iterative method would not appear well suited to the analysis of circuits with strong bilateral coupling. However, an ITA capability has been implemented in the SPLICE1 program [19], [51], and while strong bilateral coupling does increase simulation time, the correct solution is obtained even for analog circuits. With the event-driven selective trace scheduling as implemented in SPLICE1, less than a factor of two increase in CPU time has been observed compared with SPLICE1 timing simulation, for large digital circuits. For small tightly coupled MOS analog circuits, the ITA program may take even longer than SPICE2, as illustrated in the next section. However, for large integrated circuits such tight coupling is local to a small block and the advantages obtained from circuit latency, as well as the ability to exploit parallel processing effectively, far outweigh the disadvantages.

The following algorithm illustrates the principle steps involved in ITA analysis for use on a conventional computer. Only the Gauss-Seidel form is shown here, but the Gauss-Jacobi form can be obtained as outlined in Section II. At each time at which one or more nodes are scheduled to be processed, two event lists,  $E_A(t_n)$  and  $E_B(t_n)$ , are used to separate the nodes to be processed in successive iterations,  $k$  and  $k+1$ , of the Gauss-Seidel-Newton process.

*Gauss-Seidel Iteration:*

put all nodes that are connected to independent sources in event list  $E_A(0)$ :

```

 $t_n = 0;$ 
while ( $t_n < TSTOP$ ) {

     $k \leftarrow 0;$ 
    while (event list  $E_A(t_n)$  is not empty) {

        foreach ( $i$  in  $E_A(t_n)$ ) {
            obtain  $v_i^{k+1}$  from  $g_i(v_1^{k+1}, \dots, v_i^{k+1}, \dots, v_N^k) = 0$ 
            using a single Newton-Raphson step;

            if ( $|v_i^{k+1} - v_i^k| \leq \epsilon$ ; i.e. convergence is achieved) {
                use LTE to determine the nexttime,  $t_i$ , for
                processing node  $i$ ;
                add node  $i$  to event list  $E_A(t_i)$ ;
            }
            else {
                add node  $i$  to event list  $E_B(t_n)$ ; add the
                fanout nodes of node  $i$  to event list  $E_A(t_n)$  if
                they are not already on  $E_A(t_n)$ ;
            }
        }

         $E_A(t_n) \leftarrow E_B(t_n); E_B(t_n) \leftarrow \text{empty};$ 
         $k \leftarrow k + 1;$ 
    }

     $t_n \leftarrow t_{n+1};$ 
}
    
```

where  $t_n$  is the present time for processing and  $t_{n+1}$  is the next time in the time queue at which an event was scheduled. In this way, the "time step" is handled independently for each node.

This simplified algorithm does not illustrate how such issues as time-step reduction and local truncation-error estimation are handled. These and other important details of the algorithm are described elsewhere [53].

#### D. Circuit Examples

Iterated Timing Analysis is an integral part of the SPLICE1.6 mixed-mode simulator [19], [51] and a number of example runs performed by this program are included below. SPLICE1.6 uses an Successive Over Relaxation (SOR)-Newton method [29], which defaults to Gauss-Seidel-Newton for solving the nonlinear difference equations of (3). The program uses event-driven selective trace analysis to exploit latency [52]. A fundamental limitation of the present implementation of ITA, which reduces its overall effectiveness, is its simple time-step control mechanism. Another ITA program, SPLICE2 [53], is under development which overcomes this and other limitations of SPLICE1.

For large digital circuits, SPLICE1.6 is still 10–50 times faster than SPICE2 for the same output waveforms. The actual speedup factor depends on the nature of the digital circuit (highly pipelined, random logic, etc.) and the type of MOS technology in use (2-phase static, 4-phase dynamic, etc.). In each case, the circuit latency and strength of the bilateral coupling between circuit blocks determines the actual speedup. Table I contains results for the analysis of a large, random logic circuit. The corresponding output waveforms are shown in Fig. 12(a). Fig. 12(b) shows a comparison with SPICE2 for a small glitch in the waveform. Note that the program is significantly faster than SPICE2 for substantially the same waveform information. The precise tradeoff between accuracy and speed can be adjusted by varying the convergence criteria of both programs.

A major drawback of standard timing simulators is their inability to handle floating elements accurately, in particular, floating capacitors. As shown in the previous section, this is most apparent when the value of the floating capacitor is large with respect to the grounded capacitors at each of its terminals. Strong feedback and high gain also present a difficult problem for a relaxation-based ITA program. Fig. 13(a) shows the schematic diagram of an MOS operational amplifier used as part of a phase-locked loop circuit [54]. The circuit was analyzed by both SPLICE1.6 and SPICE2 in a unity-gain configuration. Such circuits have always proved the most difficult even for conventional circuit simulators. Note the large capacitive feedback provided by the floating compensation capacitor. All transistors included parasitic capacitors  $C_{gs}$  and  $C_{gd}$ . The output waveforms for both SPLICE1.6 and SPICE2, for the same step input, are shown in Fig. 13(b). The only differences in the waveforms are at the beginning of the analysis and are due to slightly different initial conditions assumed by each

TABLE I  
COMPARISONS OF CONVENTIONAL CIRCUIT SIMULATION, ITERATED  
TIMING ANALYSIS, AND TIMING SIMULATION FOR TWO EXAMPLE  
CIRCUITS

Circuit:	Encoder/decoder		Operational Amplifier	
MOSFETS:	1,326		15	
Nodes:	553		14	
	Time (s)	Memory (Kbyte)	Time (s)	Memory (Kbyte)
SPICE2G	115.640	2,420	59.8	29.0
SPLICE1.6	1.740	68.9	114.3	9.3
SPLICE1.3	789	64.4	—	—

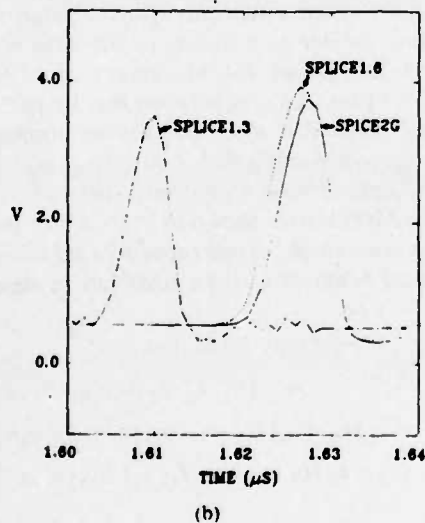
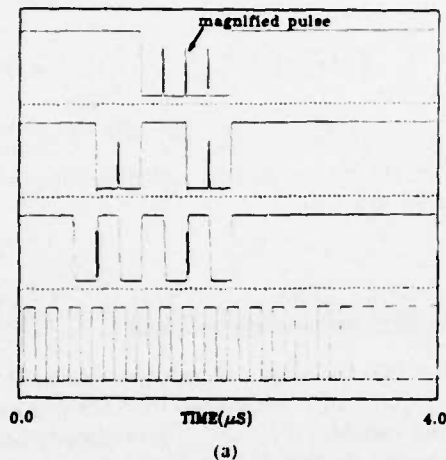


Fig. 12. (a) Selected waveforms from the encode/decode circuit obtained by SPLICE1.6. (b) Expanded view of a small pulse in the encode/decode circuit waveforms.

program. In this case, however, the prototype SPLICE1.6 program ran two times longer than SPICE2, as shown in Table I. While it is to be expected that such a worst-case circuit would reduce the performance of the program due to the strong capacitive feedback and high forward gain of the circuit, it is anticipated that this time difference will be reduced as the SPLICE1 program is developed further.

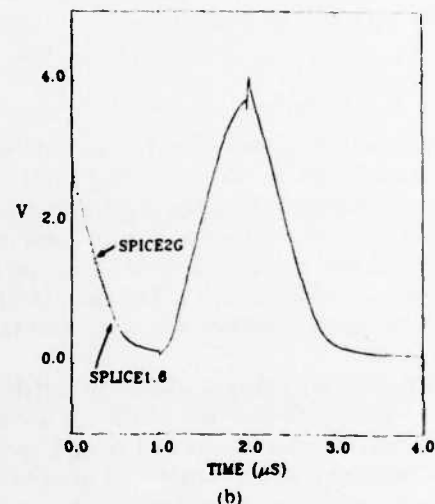
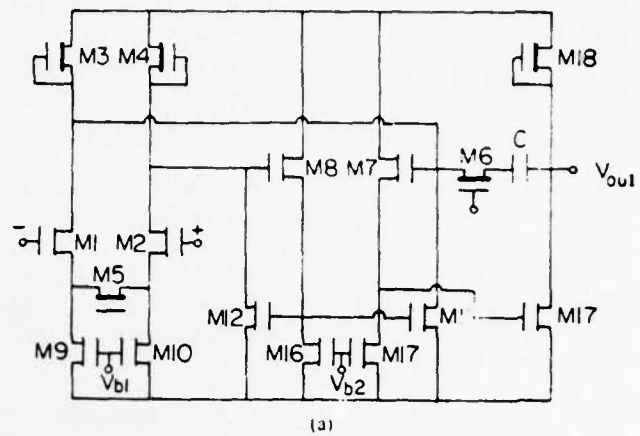


Fig. 13. (a) Schematic diagram of operational amplifier circuit. (b) Response of operational amplifier to pulse input.

### E. Conclusions

As previously illustrated, Iterated Timing Analysis has a great deal of potential for improving the performance of electrical simulation. Not only can this technique outperform standard circuit-simulation programs for the analysis of large circuits on standard computers, but it offers the possibility of a further substantial speedup when implemented on special-purpose hardware.

## V. WAVEFORM RELAXATION TECHNIQUES

### A. Introduction

Both timing algorithms and iterated timing analysis are based on the application of relaxation techniques to the solution of circuit equations at the nonlinear algebraic equation level. As pointed out in Section II, there is a part missing in Fig. 3: a relaxation method at the differential equation level. While relaxation techniques in the linear and nonlinear algebraic case deal with vectors in  $\mathbb{R}^n$ , relaxation techniques at the differential equation level must deal with elements in function spaces, i.e., waveforms. Recently, a family of relaxation techniques applied to the differential equation level, called *Waveform Relaxation*

(WR), has been proposed in [20], [21]. WR methods have been implemented in an experimental circuit simulator called *RELAX* [21] that has proven to be effective for the accurate analysis of some MOS digital circuits with more than an order of magnitude speed improvement over standard circuit simulators.

In this Section, the basic ideas of WR methods are reviewed and some of WR applications and extensions are presented. To begin, a simple example is used to illustrate the method, and then the general "Gauss-Seidel" algorithm in the WR family for MOS digital circuits is described. A more detailed and complete description of these techniques is available in [20], [21].

### B. The Waveform-Relaxation Gauss-Seidel Algorithm

Consider the first-order two-dimensional differential equation in:  $x(t) \in \mathbb{R}^2$  on  $t \in [0, T]$ .

$$\dot{x}_1 = f_1(x_1, x_2, t), \quad x_1(0) = x_{10} \quad (44a)$$

$$\dot{x}_2 = f_2(x_1, x_2, t), \quad x_2(0) = x_{20} \quad (44b)$$

The basic idea of the "Gauss-Seidel" waveform-relaxation algorithm is to fix the waveform  $x_2: [0, T] \rightarrow \mathbb{R}$  and solve (44a) as a one-dimensional differential equation in  $x_1(\cdot)$ . The solution thus obtained for  $x_1$  can be substituted into (44b) which will then reduce to another first-order differential equation in one variable,  $x_2$ . Equation (44a) is then resolved using the new solution for  $x_2(t)$ , and the procedure is repeated.

In this fashion, an iterative algorithm has been constructed. It replaces the problem of solving a differential equation in two variables by one of solving a sequence of differential equations in one variable. As described earlier, the waveform relaxation algorithm can be seen as an analogue of the Gauss-Seidel technique for solving nonlinear algebraic equations. Here, however, the unknowns are waveforms (elements of a function space), rather than real variables. In this sense, the algorithm is a technique for time-domain decoupling of differential equations.

WR algorithms applied to circuits can be formulated in a number of ways. A "Gauss-Seidel" WR algorithm for MOS circuits will be considered in the following analysis. Recall that, according to (2), the circuit equations are formulated as

$$C(v, u)\dot{v} + f(v, u) = 0, \quad v(0) = V \quad (45)$$

where  $C: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^{n \times n}$  is a symmetric diagonally dominant matrix-value function in which  $-C_{ij}(v, u); i \neq j$  is the total floating capacitance between nodes  $i$  and  $j$ .  $C_{ii}(v, u)$  is the sum of the capacitances of all capacitors connected to node  $i$ , and  $f: \mathbb{R}^n \times \mathbb{R}^r \rightarrow \mathbb{R}^n$  is a continuous function, each component of which represents the net current charging the capacitor at each node due to the pass transistors, the other conductive elements, and the controlled current sources.

*Algorithm V-B-1 (WR Gauss-Seidel Algorithm for Solving (45)):* Comment:

The superscript  $k$  denotes the iteration count, the subscript  $i$  denotes the component index of a vector, and  $\epsilon$  is a

small positive number.

$k \leftarrow 0$ ;

guess waveform  $v^0(t); t \in [0, T]$  such that  $v^0(0) = V$   
(for example, set  $v^0(t) = V, t \in [0, T]$ );

repeat {

$k \leftarrow k + 1$

foreach ( $i$  in  $N$ ) {

solve

$$\sum_{j=1}^i C_{ij}(v_1^k, \dots, v_i^k, v_{i+1}^{k-1}, \dots, v_N^{k-1}, u) \dot{v}_i^k +$$

$$\sum_{j=i+1}^n C_{ij}(v_1^k, \dots, v_i^k, v_{i+1}^{k-1}, \dots, v_N^{k-1}, u) \dot{v}_i^{k-1} +$$

$$f_i(v_1^k, \dots, v_i^k, v_{i+1}^{k-1}, \dots, v_N^{k-1}, u) = 0$$

for  $(v_i^k(t); t \in [0, T])$ , with the initial condition  $v_i^k(0) = V_i$ .

}

} until  $(\max_{1 \leq i \leq n} \max_{t \in [0, T]} |v_i^k(t) - v_i^{k-1}(t)| \leq \epsilon)$   
that is, until the iteration converges. ■

Note that (45) has only one unknown variable  $v_i^k$ . The variables  $v_{i+1}^{k-1}, \dots, v_N^{k-1}$  are known from the previous iteration and the variables  $v_1^k, \dots, v_i^{k-1}$  have already been computed. Note also that the Gauss-Jacobi version of the WR algorithm presented earlier can be obtained simply by replacing the **foreach** statement with the **forall** statement and adjusting the iteration indices in the same way as can be done in the Gauss-Jacobi version of ITA. In the Gauss-Jacobi case, the computation can be performed in parallel, and hence it is more suitable for implementation on special-purpose hardware.

As an example of how Algorithm V-B-1 can be applied, consider the MOS circuit shown in Fig. 14. For the sake of simplicity it is assumed that all capacitors are linear. Hence the dynamical behavior of the circuit can be described as follows:

$$\begin{aligned} (c_1 + c_2 + c_3)\dot{v}_1 - i_1(v_1) + i_2(v_1, u_1) \\ + i_3(v_1, u_2, v_2) - c_1\dot{u}_1 - c_3\dot{u}_2 = 0 \\ (c_4 + c_5 + c_6)\dot{v}_2 - c_6(\dot{v}_3) - i_3(v_1, u_2, v_2) - c_4\dot{u}_2 = 0 \\ (c_6 - c_7)\dot{v}_3 - c_6\dot{v}_2 - i_4(v_3) + i_5(v_3, v_2) = 0. \end{aligned} \quad (46)$$

Applying the WR procedure to (46) the  $k$ th iteration corresponds to solving the following equations:

$$\begin{aligned} (c_1 + c_2 + c_3)\dot{v}_1^k - i_1(v_1^k) + i_2(v_1^k, u_1) \\ + i_3(v_1^k, u_2, v_2^{k-1}) - c_1\dot{u}_1 - c_3\dot{u}_2 = 0 \\ (c_4 + c_5 + c_6)\dot{v}_2^k - c_6\dot{v}_3^{k-1} - i_3(v_1^k, u_2, v_2^{k-1}) - c_4\dot{u}_2 = 0 \\ (c_6 + c_7)\dot{v}_3^k - c_6\dot{v}_2^k - i_4(v_3^k) + i_5(v_3^k, v_2^k) = 0 \end{aligned} \quad (47)$$

The circuit interpretation of (47) is shown in Fig. 15.



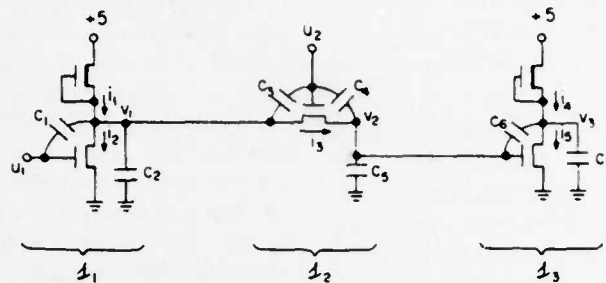


Fig. 14. Circuit example for Gauss-Seidel waveform relaxation algorithm.

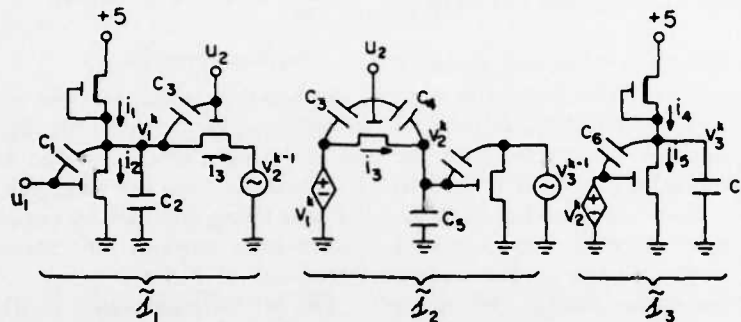


Fig. 15. Circuit interpretation of Gauss-Seidel waveform relaxation applied to the circuit in Fig. 14.

If the original circuit in Fig. 14 consists of 3 subcircuits  $s_1$ ,  $s_2$ , and  $s_3$ , then the decomposed subcircuits  $\tilde{s}_1$ ,  $\tilde{s}_2$ , and  $\tilde{s}_3$  together with additional components to approximate the loading effects due to the rest of the circuit. Hence, the WR procedure for analyzing the circuit in Fig. 14 can be described in circuit terms as follows:

$k \leftarrow 0$ ;

make an initial guess of  $v_2^0(t)$ ,  $v_3^0(t)$ ;  $t \in [0, T]$ ;

repeat {

$k \leftarrow kp$ ;

analyze  $s_1$  for its output waveform  $v_1^k(\cdot)$  by approximating the loading effect due to  $s_2$ ;

analyze  $s_2$  for its output waveform  $v_2^k(\cdot)$  by using  $v_1^k(\cdot)$  as its input and approximating the loading effect due to  $s_3$ ;

analyze  $s_3$  for its output waveform  $v_3^k(\cdot)$  by using  $v_2^k(\cdot)$  as its input;

}

until (the difference between  $\{(v_1^k(t), v_2^k(t), v_3^k(t)); t \in [0, T]\}$  and  $\{(v_1^{k-1}(t), v_2^{k-1}(t), v_3^{k-1}(t)); t \in [0, T]\}$  is sufficiently small) ■

From the previous procedure and example it can be seen that each component of the decomposition is a dynamical subcircuit which is processed for the entire time interval  $[0, T]$  in a fixed order. When each subcircuit is being processed, the iterations (coupling or loads) from the rest of the circuit are approximated by using the information obtained from the most recent iteration. The iteration is carried out until satisfactory convergence of all waveforms

is detected. It can be shown that for the MOS circuit in Fig. 14 the sequence of waveforms generated by the WR procedure will always converge to the correct waveform independent of the initial guess provided that  $c_2$ ,  $c_3$ , and  $c_7$  are not zero. In [20], [50], a strong convergence result was proved that can be applied to Algorithm V-B-1 as follows.

*Theorem V-B-1:* Assume that:

1) the charge-voltage characteristic of each capacitor, the current-voltage characteristic of each conductor, and the drain-current characteristic of each MOS device are Lipschitz continuous with respect to their controlling variables;

2)  $C_{\min} > 0$  and  $C_{\max} < \infty$  where  $C_{\min} \in \mathbb{R}$  is the minimum value of all grounded capacitors at any permissible value of node voltage, and  $C_{\max} \in \mathbb{R}$  is the maximum value of all floating capacitors at any permissible value of node voltages; and

3) the current through any controlled conductor (e.g., the drain current of an MOS device) is uniformly bounded throughout the relaxation process.

Then, for any given set of initial conditions and any given piecewise continuous input  $u(\cdot)$ , Algorithm V-B-1 generates a converging sequence of iterated solutions whose limit satisfies the circuit equations and the given initial conditions. ■

### C. Waveform Relaxation in RELAX

The WR "Gauss-Seidel" algorithm was implemented in an experimental circuit simulator, RELAX. Actually, RELAX implemented a modified version of the WR algorithm described previously. These modifications were intro-

duced to improve the speed of convergence of the algorithm and exploit the structure of the class of circuits to be analyzed, i.e., MOS digital circuits. These modifications are as follows.

1) Rather than having strictly one unknown per each component of the decomposition as stated in Algorithm V-B-1, RELAX allows each decomposed subcircuit to have more than one unknown. This corresponds to a block relaxation method that can be proven to have similar convergence properties. In fact, the analysis of the circuit is decomposed into the analysis of subcircuits each of which corresponds to a physical subcircuit that is built into the program and called by the user as a unit, such as a NOR or a NAND.

2) Each decomposed subcircuit is processed by using standard circuit-analysis techniques. The Backward Euler integration method with variable time steps is used to discretize the differential equations associated with the subcircuit, and the Newton-Raphson method is used to solve the nonlinear algebraic equations resulting from the discretization. Since the number of unknowns associated with a subcircuit is usually small, the linear equation solver used by the Newton-Raphson method is implemented by using standard full-matrix techniques rather than using sparse-matrix techniques. Note that in RELAX each subcircuit is analyzed independently from  $t = 0$  to  $t = T$ , using its own time-step sequence, controlled by the integration method, whereas in a standard circuit simulator the entire circuit is analyzed from  $t = 0$  to  $t = T$  using only one common time-step sequence. In RELAX, the time-step sequence of one subcircuit is usually different from the others, but contains, in general, a smaller number of time steps than that used in a standard circuit simulator for analyzing the same circuit.

3) The order according to which each subcircuit is processed is determined in RELAX prior to starting the iteration by a subroutine called "scheduler." Although, according to Theorem V-B-1, scheduling is not necessary to guarantee convergence of the iteration, it does have an impact on the speed of convergence as is the case for the relaxation methods at the linear- and nonlinear-equation levels. Assume now that the circuit consists of unidirectional subcircuits with no feedback path. Exactly as for the other relaxation methods introduced before, if the subcircuits are processed according to the flow of signals in the circuit, the algorithm used in RELAX will converge in just two iterations (actually the second iteration is needed only to verify that convergence has been obtained). For MOS digital circuits which contain almost unidirectional subcircuits, it is intuitive that convergence of the WR procedure will be achieved more rapidly if the subcircuits are processed according to the flow of signals in the circuit. The scheduler traces the flow of signals through the circuit and generates a static order for the processing of subcircuits. To be able to trace the flow of signals, the scheduler requires the user to specify the flow of signals through each subcircuit by partitioning the terminal of the subcircuit into input and output terminals. This is needed since

in RELAX the basic unit is a subcircuit. In general, a designer can easily specify what the flow of the signals is intended to be even in a subcircuit which is not unidirectional such as a transmission gate or a subcircuit containing floating capacitors between its input and output terminals. The analysis algorithm in RELAX will indeed take into account the bidirectional effects correctly.

4) The first iteration in RELAX is carried out by assuming that there is no loading effect due to fanouts. The "standard" WR procedure actually begins at the second iteration in RELAX. Hence, strictly speaking, the first iteration in RELAX is used to generate a good initial guess for the actual WR procedure.

#### D. Speed-up Techniques

In addition to the previous modifications, RELAX incorporates two bypass techniques to speed up the process of analyzing a subcircuit. The key idea is once more to bypass the analysis of a subcircuit for certain time intervals without losing accuracy by exploiting the information obtained from previous time points and/or from previous iterations.

The two techniques used in RELAX are presented here by showing their application for the analysis of the subcircuit  $s_1$  of the circuit shown in Fig. 16, which is a schematic diagram of the circuit in Fig. 14. The output voltages of  $s_1$  and  $s_2$  at the  $k$ th iteration are denoted by  $v_1^k$  and  $v_2^k$ , respectively.

The first technique is based on the latency of  $s_1$  and is similar to the technique described in Section III. According to (47),  $s_1$  is analyzed in the first iteration with no loading effect from  $s_2$ . After it has been analyzed for a few time points, its output voltage  $v_1^1$  is found to be almost constant with time, i.e.,  $v_1^1(0.01) \approx 0$  (see Fig. 16(b)). Since the input of  $s_1$ , i.e.,  $u_1$  is also constant during the interval  $[0.01, 1.9]$ , the subcircuit  $s_1$  is said to be "latent" in the first iteration during the interval  $[0.01, 1.9]$ , and its analysis during this interval is bypassed. From Fig. 16(b),  $s_1$  is latent again in the interval  $[2.15, 3]$ . Note that, according to (47), the check for latency of  $s_1$  after the first iteration will include  $u_2$  and  $v_2$  as well as  $u_1$  since they can affect the value of  $v_1$ . For most digital circuits, the latency intervals of a subcircuit usually cover a large portion of the entire simulation time interval  $[0, T]$  and hence the implementation of this technique can provide a considerable saving of computing time as shown in Table II.

The second technique is based on the *partial convergence* of a waveform during the previous two iterations. This technique is introduced by using the example of Fig. 16. After the first two iterations, we observe that the values of  $v_1^1$  and  $v_1^2$  during the interval  $[1.7, 3.0]$  do not differ significantly (see Fig. 16(b), (c)), i.e., the sequence of waveforms of  $v_1$  seem to converge in this interval after two iterations. In the third iteration, shown in Fig. 16(d),  $s_1$  is analyzed from  $t = 0$  to  $t = 1.8$  and  $v_1^3(1.8)$  is found to be almost the same as  $v_1^2(1.8)$ . Moreover, during the interval  $[1.8, 3]$ , the value of  $v_2^3$  which affects the value of  $v_1^3$  also does not differ significantly from the values of  $v_2^2$  (which

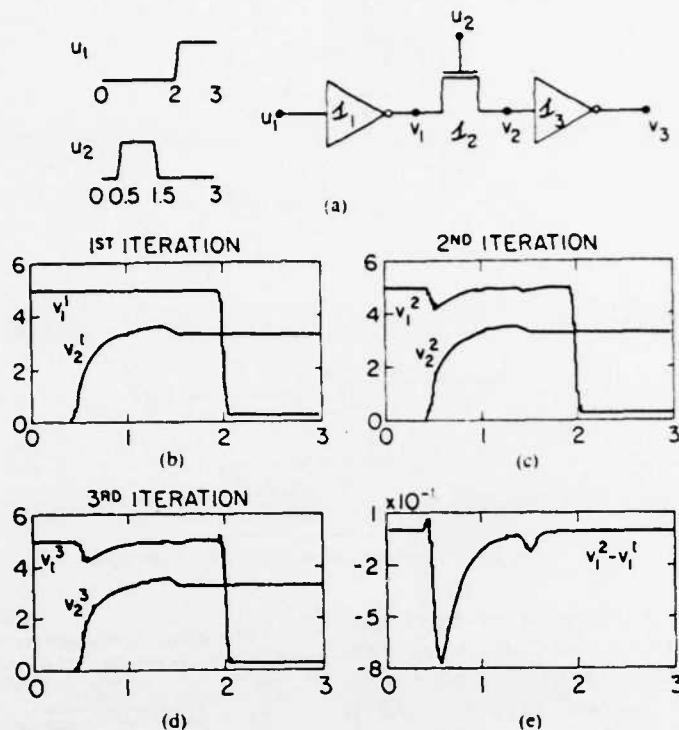


Fig. 16. (a) Schematic diagram of the circuit of Fig. 14. (b), (c), and (d): The waveforms of the two node voltages at the first, second and third waveform relaxation iteration, respectively. (e): The difference of the waveforms between the first and second iteration.

affects the value of  $v_1^2$ ). Hence the value of  $v_1^3$  during the interval  $[1.8, 3]$  should remain the same as  $v_1^2$  and the analysis of  $s_1$  during this interval in the third iteration will be bypassed. This technique can provide a considerable saving of computing time as shown in Table II since the intervals of convergence can cover a large portion of the entire simulation time interval  $[0, T]$ , especially in the last few iterations. Note that the subcircuit need not be latent during the intervals of convergence although overlapping of these intervals with the latency intervals is possible.

#### E. Some Extensions of Waveform-Relaxation Techniques: RELAX2

RELAX is written in FORTRAN77. It can handle MOS digital circuits containing NOR gates, NAND gates, transmission gates, multiplexers (or banks of transmission gates whose outputs are connected together), super buffers, and cross-coupled NOR gates (or RS flip-flops). It uses the Shichman-Hodges model [33] for the MOS device. All the computations were performed in double precision and the results were also stored in double precision. Although the RELAX code is rather small, approximately 4000 FORTRAN lines, it requires a large amount of storage for the waveforms, especially when large circuits are analyzed. For an MOS circuit containing 1000 nodes with 100 analysis time points per node, the waveform storage requires approximately  $3 \times 1000 \times 1000$  floating point numbers (corresponding to 2.4 Mbytes if each number is stored in 64 bits).

TABLE II  
COMPARISON OF CPU TIMES USED BY RELAX FOR ANALYZING THE CIRCUIT OF FIG. 0 WITH AND WITHOUT THE LATENCY AND THE PARTIAL WAVEFORM CONVERGENCE TECHNIQUES

(Case 1: Without the latency and the partial waveform convergence techniques.  
Case 2: With only the latency technique.  
Case 3: With only the partial waveform convergence technique.  
Case 4: With both the latency and the partial waveform convergence techniques.)

Iteration #	CPU time ( seconds )			
	case 1	case 2	case 3	case 4
1	0.363	0.255	0.360	0.258
2	0.624	0.704	0.814	0.695
3	0.621	0.705	0.842	0.738
4	0.625	0.704	0.181	0.104
5	0.632	0.695	0.097	0.016
Total	3.666	3.063	1.864	1.311

A new version of RELAX, RELAX2 [55], written in C, has extended and made waveform-relaxation algorithms more practical. The first important extension consists of allowing arbitrary subcircuits to be defined by the user, as was provided in the original SPLICE1 program. These

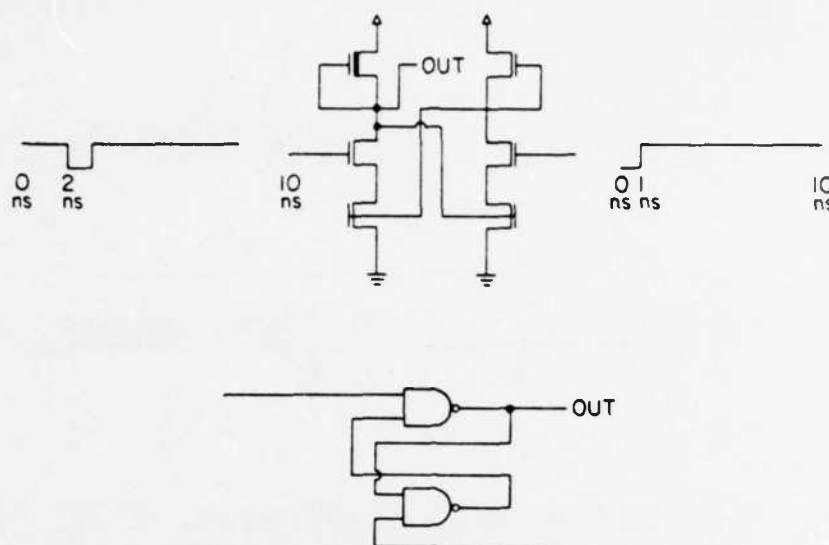


Fig. 17. A two cross-coupled NAND-gate circuit.

subcircuits are analyzed by a subroutine patterned after SPICE, while the original RELAX used "hard-wired" subcircuit analyzers made possible by the fixed structure of the subcircuits allowed by the program. Thus RELAX2 is slower than RELAX, but still maintains a definite advantage over standard circuit simulators, approximately one order of magnitude speed improvement.

The second extension consists of allowing the decomposition of the time interval of interest for the time-domain simulation into subintervals, called *windows* [56].

Digital circuits can be broken up into two very broad classes: circuits with logic feedback loops (finite-state machines, asynchronous circuits, digital oscillators) and circuits without logic feedback loops (most combinational logic, programmable logic arrays). Experience simulating MOS digital circuits using RELAX2 shows that most MOS digital circuits without logic feedback loops converge in less than ten iterations. However, circuits with logic feedback loops may take many more iterations to converge, and the number of iterations required is proportional to the length of the simulation interval.

This suggests that the interval of simulation should be broken into "windows",  $[0, T_1]$ ,  $[T_1, T_2]$ ,  $\dots$ ,  $[T_{n-1}, T_n]$ , so that the relaxation will converge rapidly in each window. Waveform relaxation is applied to the first window,  $[0, T_1]$  and the values of the node voltages at  $T_1$  are then used as initial conditions for the analysis of the second window. This procedure is repeated until all windows have been analyzed.

Consider the analysis of the cross-coupled NAND gate circuit shown in Fig. 17 with the "window" approach provided by RELAX2; convergence is quite rapid (see Table III). There is a trade-off, however. As the window size gets smaller some of the advantages of waveform relaxation are lost. One cannot take advantage of a digital circuit's natural latency over the entire waveform, but only within that window. The scheduling overhead increases when the windows become smaller, as each subcircuit must

TABLE III  
WINDOWING EXPERIMENTS FOR THE CROSS-COUPLED NAND GATES PERFORMED ON A VAX11/780 RUNNING BERKELEY UNIX4.1c

CROSS-COUPLED NAND GATES			
# Windows	# Timepoints	Max # Iterations	CPU time*
SPICE	—	—	21.75
1	50	>100	—
2	50	10	13.59
4	48	4	4.48
8	59	4	5.36
16	89	4	5.91

be scheduled once for each window, and if the windows are made very small, time steps chosen to calculate the waveforms will be limited by the window size rather than by the local truncation error and unnecessary calculations will be performed.

Breaking the interval simulation into windows also has the advantage of reducing the memory requirements of WR, since these are proportional to the number of time points used by the numerical integration algorithm used to solve the subcircuit equations.

Other extensions to WR included in RELAX2 involve the approximate solution of the subcircuit equations [56], [57]. When using iterative decomposition methods for solving systems of nonlinear equations, it may be possible to reduce the calculations required by not solving the decomposed nonlinear equations exactly at each iteration. In some cases the convergence of the algorithm is not affected by the inaccurate solutions. An example is the Gauss-Seidel-Newton method [27] described in Section II. In the WR case, simpler approximate methods for calculating the node waveforms are used for the first few iterations; more complex and more exact methods are used for the last few iterations.

One way of simplifying the calculation of the node-voltage waveforms is to use a simple model for the MOS devices and then to switch to the more detailed model as the waveforms approach convergence. The simple device model used in RELAX2 is a resistor in series with a switch,



where the size of the resistance is scaled with the device size. Using such a model in the calculation of waveforms is not straightforward, because the equations describing the model can not be solved easily using the Newton-Raphson method. The Newton-Raphson method will often oscillate about the point where the simple model's switch changes state. One solution to this problem is not to carry out the Newton-Raphson method to convergence, but to do only one iteration. The result is that the calculation of the waveforms using the simple model is quite fast, but only approximate, even if the simple model is assumed to be correct. The results obtained using the simple model and then changing to the more detailed model have been disappointing so far, as demonstrated in the examples of Table IV. In circuits without logic feedback, the simple model did not provide a better guess for the waveforms than one iteration using more complex models. It is possible that the addition of another term to the simple model, to make it more smooth, may help. Then the Newton-Raphson algorithm can be used and achieve the accuracy required to produce a useful first guess for the iterations using the more detailed models.

Another approach to simplifying the calculations performed in the first few iterations of the WR algorithm is to allow the numerical integration algorithm, which is used to solve for the node waveforms of the decomposed subcircuits, to use a larger local truncation error. Here, unlike changing the device models, it is possible to increase the accuracy of the calculation of the node waveforms at each iteration by tightening the local truncation error limit. In the case of RELAX, since most circuits converge in about 5 iterations, a local truncation error was chosen that is about 3 times larger than the local truncation error that would be chosen to calculate the waveforms for the final answer. Then after each iteration the local truncation error is multiplied by 0.7. The results from this approach are shown in the last row of Table IV.

A key question to be asked is whether the convergence of WR algorithms is affected by these approximation techniques. Note that Theorem V-B-1 assumes that the solutions of the differential equations are computed exactly. However, the convergence of the WR algorithm to the exact solution of the original differential equations has been proven provided the accuracy of the integration is increased while the WR iterations are converging [56]. The framework necessary to prove this result is the one of nonstationary WR algorithms.

Algorithm V-B-1 is a stationary algorithm in the sense that the iteration process is performed with the same set of equations. Nonstationary WR algorithms are characterized by the fact that the equations describing the system at each iteration can change from one iteration to the other. Note that the approximations computed by the integration methods can be viewed as the exact solutions of perturbed differential equations. Thus the iteration equations of the WR algorithms can be seen as changing from iteration to iteration. The convergence theorem proven in [50], [56], assumes that the accuracy of the integration methods is

TABLE IV  
EXPERIMENTS WITH VARIABLE ACCURACY MODEL AND VARIABLE LOCAL TRUNCATION ERROR (LTE)

Method	TEST CIRCUITS					
	Shift Cell		Two Phase Clk		Memory Cell	
	# Iter	Time	# Iter	Time	# Iter	Time
SPICE	—	12.52	—	43.13	—	13.63
RELAX2	4	2.10	4	5.47	4	2.98
Simple model	4	3.20	4	8.75	4	4.45
Simple model only	4	0.49	4	0.69	4	0.88
LTE	5	1.24	3	3.81	4	2.71

controllable and that in the limit, the exact solution of a differential equation is achieved. Fortunately, this property is obviously satisfied when using simpler models in the first iterations to switch to more complex and accurate models at the last iterations. In addition, it is known that consistent integration methods can be made as accurate as desired by controlling parameters such as the local truncation error. Thus the speed-up techniques presented here are theoretically sound.

### F. Conclusions

Waveform-relaxation methods have been proven to be effective decomposition methods for the analysis of large-scale MOS circuits. In particular, the methods have guaranteed convergence properties. Since WR algorithms are quite new, more research is needed to characterize completely the trade-offs involved in the choice of a particular method in the class (e.g., Gauss-Jacobi versus Gauss-Seidel). In addition, an accurate comparison between iterated timing analysis and waveform relaxation must be carried out.

It is clear that WR methods are quite suitable for implementation on a parallel or pipeline architecture since they allow different subcircuits to be analyzed concurrently on different processors.

In addition, WR algorithms have recently been extended to piecewise linear circuit analysis [57] and to other fields such as electrophoresis process simulation [58].

## VI. SUMMARY AND DIRECTIONS FOR FUTURE WORK

### A. Introduction

Relaxation-based simulation techniques have been used for the analysis of electronic circuits in many ways. Timing simulators were the first relaxation-based electrical simulators to gain widespread use and are still being used successfully in many companies today. Unfortunately, since only a single sweep of a relaxation method is used to approximate the solution of the set of nonlinear algebraic equations obtained at a time point, these simulators suffer from severe accuracy problems when used to analyze circuits containing tight feedback loops or floating circuit elements. In Section III we presented an analysis of the numerical techniques used in timing simulation, and described a number of algorithms which can be used to improve the accuracy of timing simulators. In particular, methods suited to the analysis of floating capacitors have been described.

While timing simulators will continue to be used for the analysis of circuits where constrained circuit design meth-

ods, such as cell-based approaches, limit the likelihood of simulation errors, they cannot be used for the analysis of complex digital and analog circuits where feedback effects are significant. A new relaxation-based approach, called iterated timing analysis, can be used for the analysis of these circuits. We have described the basic algorithms used for ITA and their associated numerical properties in Section IV. This approach provides accurate simulation results while still achieving a substantial speed improvement over conventional circuit-analysis techniques. As shown in Section IV, ITA has also been used successfully for the analysis of tightly coupled analog circuits containing a number of floating capacitors.

Both of the aforementioned techniques apply relaxation methods to the solution of a set of nonlinear algebraic equations. In contrast to these techniques, the waveform-relaxation method uses a relaxation approach at the differential equation level. In Section V we have shown that waveform relaxation has guaranteed convergence properties for a wide class of electrical circuits, and has performed over one order of magnitude faster than standard circuit simulators on a number of test circuits while maintaining the same, or even better, accuracy. A number of improvements and extensions to the basic waveform-relaxation method were also presented.

On conventional computers, the speed advantage of relaxation-based analysis over matrix-based techniques can vary from a slight slow-down, for small tightly coupled analog circuits, to a maximum of about two orders of magnitude speedup, for large semistatic digital circuits.

### B. Special-Purpose Hardware

The use of special-purpose computer instructions for sparse-matrix solution [14] and the use of vector computers, such as the CRAY1 [17], can improve the speed of conventional circuit simulators by about an order of magnitude over their nonoptimized versions on the same machine. In the latter case, the speedup is limited by the *gather/scatter* problem [46] associated with arranging the data so that effective parallel computation can be performed. Unfortunately, the irregular structure of a circuit sparse matrix is the limiting factor here.

If relaxation techniques are used to replace these direct methods, the solution of each node equation is effectively *decoupled* from the others. While such decoupled-analysis techniques would be suitable for use on a vector computer, it seems that other architectures, in particular *data-flow* computers and related dependency-driven approaches [48], [49], [59]–[62], will allow the decoupling to be exploited more effectively. A straight-forward approach to the implementation of an electrical relaxation simulator on such a computer would be to allocate a separate processor for the solution of each decoupled-node equation. For an ITA algorithm, the calculation performed by each processor would be a single Newton–Raphson step on a nonlinear algebraic equation in one unknown. In the case of WR, each calculation would involve the computation of a partial waveform, or set of waveforms, on the processor. While the

performance of a practical multiprocessor depends on many factors, a simplified analysis is presented here to illustrate the potential savings of such a machine.

For a circuit containing  $N$  nodes with  $M$  nodes actively changing at any time,  $M \leq N$ , the total time spent solving the independent node equations on a serial computer is approximately

$$T_s(M) \propto Mt_s \quad (48)$$

where  $t_s$  is the time required to solve the single-node equation in either scheme. Consider a multiprocessor using a single- or multiple-stage shuffle network [61], [62] with an element cycle time of  $t_c$  and a latency proportional to  $k \log(P)$ , where  $P$  is the number of ports in the shuffle network and  $k$  is a constant. For now assume  $P > M$ ; then the total analysis time on such a network is approximately

$$T_p(P) \propto t_s + t_c k \log(P). \quad (49)$$

Equation (49) is in fact a worst-case figure because it assumes all communication is on the critical path of the computation and that there is no pipelining of requests. The speed-up factor for the parallel computation is then

$$\frac{T_s}{T_p} \propto \frac{Mt_s}{t_s + t_c k \log(P)}. \quad (50)$$

If  $k$  is from 1 to 3 [63]; if  $M = P$ ; if  $t_c \approx t_s$ , the speed-up becomes approximately  $M/\log M$ . However, if the equation solution time is larger than the network cycle time, or if a better than random placement of the circuit nodes on the network can be performed, then the speed-up factor will be closer to  $M$ . Note that it will generally be true in practice that  $M > P$ . In that case, more than one node will be allocated to each processor. Techniques for the implementation of “virtual processors” can be used to solve this problem [64], and scheduling algorithms can be used to allocate the nodes to processors in such a way that network loading is uniform.

It is clear that relaxation-based algorithms for electrical simulation are well suited to the use of special-purpose hardware. Future work in this area includes the investigation of the best match of hardware and algorithms, and the investigation of optimal techniques for simulation time advancement in a parallel computational environment.

### ACKNOWLEDGMENT

The authors wish to thank the many talented graduate students with whom it has been their pleasure to work. M.-Y. Hsueh, J. E. Kleckner, J. D. Crawford, and J. Straus participated in the early development of the MOTIS-C timing simulator and SPLICE1 mixed-mode simulator. N. DeMicheli contributed in the development of new algorithms for timing simulation. E. Lelarsmee played a key role in the development of the waveform-relaxation techniques and the RELAX simulator. J. Kaye extended WR algorithms to the analysis of piecewise linear circuits. J. E. Kleckner and R. Saleh implemented the iterated timing analysis algorithm in SPLICE1.6. J. White wrote the RELAX2 program.

A. Ruehli participated in the early development of waveform relaxation. The authors also thank G. D. Hachtel for helpful discussions and D. O. Pederson for his continuous support, encouragement, and inspiration.

Many people have helped during the preparation of this paper. In particular, the authors thank N. DeMicheli, R. Saleh, and J. White for their assistance in the preparation of examples and critical reading of the manuscript. They would also especially like to thank J. T. Deutsch and J. E. Kleckner for helpful discussions. They would also like to thank Digital Equipment Corporation, IBM, Harris Semiconductors, Hewlett-Packard, and Tektronix for their support throughout the course of this work.

#### REFERENCES

- [1] W. Nagel, "SPICE2, A computer program to simulate semiconductor circuits," Univ. of California, Berkeley, Memo No. ERL-M520, May 1975.
- [2] "Advanced statistical analysis program (ASTAP)," IBM Corp. Data Proc. Div., White Plains, NY, Pub. No. SH20-1118-0.
- [3] A. Sangiovanni-Vincentelli, "Circuit simulation," in *Computer Design Aids for VLSI Circuits*, P. Antognetti, D. O. Pederson, and H. De Man, Eds. Groningen, The Netherlands: Sijthoff and Noordhoff, 1981, pp. 19-113.
- [4] F. Jenkins, "ILOGS: User's manual," Simutec, 1982.
- [5] S. A. Szygenda and E. W. Thompson, "Digital logic simulation in a time-based, table-driven environment. Part 1. Design verification," *Comput.*, March 1975, pp. 24-36.
- [6] R. E. Bryant, "An algorithm for MOS logic simulation," *LAMBDA*, 4th quarter, pp. 46-53, 1980.
- [7] C. M. Baker and C. Terman, "Tools for verifying integrated circuit designs," *LAMBDA*, 4th quarter 1980.
- [8] M. H. Heydemann, G. D. Hachtel, and M. Lightner, "Implementation issues in multiple delay switch level simulation," in *Proc. 1982 Int. Conf. Circ. Comp.*, pp. 46-52, Sept. 1982.
- [9] B. R. Chawla, H. K. Gummel, and P. Kozak, "MOTIS-an MOS timing simulator," *IEEE Trans. Circuits Syst.*, vol. CAS-22, pp. 901-909, Dec. 1975.
- [10] S. P. Fan, M. Y. Hsueh, A. R. Newton, and D. O. Pederson, "MOTIS-C A new circuit simulator for MOS LSI circuits," in *Proc. IEEE Int. Symp. Circuits Syst.*, Apr. 1977.
- [11] N. Tanabe, H. Nakamura, and K. Kawakita, "MOSTAP: An MOS circuit simulator for LSI," in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 1035-1039, Apr. 1980.
- [12] A. R. Newton, "Timing, logic and mixed-mode simulation for large MOS integrated circuits" in *Computer Design Aids for VLSI Circuits*, P. Antognetti, D. O. Pederson, and H. De Man, Eds. Groningen, The Netherlands: Sijthoff and Noordhoff, 1981, pp. 175-240.
- [13] J. L. Burns, A. R. Newton, and D. O. Pederson, "Active device table look-up models for circuit simulation," in *Proc. 1983 Int. Symp. Circuits Syst.*, May 1983.
- [14] E. Cohen, "Performance limits of integrated circuit simulation on a dedicated minicomputer system," ERL Memo. UCB/ERL M81/29, May 22, 1981.
- [15] A. Sangiovanni-Vincentelli, L. K. Chen, and L. O. Chua, "A new tearing approach-node tearing nodal analysis," in *Proc. IEEE Int. Symp. Circuits Syst.*, pp. 143-147, 1977.
- [16] P. Yang, I. N. Hajj, and T. N. Trick, "SLATE: A circuit simulation program with latency exploitation and node tearing," in *Proc. IEEE Int. Conf. Circuits Comput.*, Oct. 1980.
- [17] A. Vladimirescu and D. O. Pederson, "Performance limits of the CLASSIE circuit simulation program," in *Proc. Int. Symp. Circuits Syst.*, May 1982.
- [18] G. D. Hachtel and A. L. Sangiovanni-Vincentelli, "A survey of third-generation simulation techniques," *Proc. IEEE*, vol. 69, no. 10, Oct. 1981.
- [19] J. Kleckner, R. Saleh, and A. R. Newton, "Electrical consistency in schematic simulation," in *Proc. IEEE Int. Conf. Circuits Comput.*, pp. 30-34, Oct. 1982.
- [20] E. Lelarsmee, A. Ruehli, and A. L. Sangiovanni-Vincentelli, "The waveform relaxation method for the time-domain analysis of large scale integrated circuits," *IEEE Trans. Computer-Aided Design of ICAS*, vol. CAD-1, no. 3, pp. 131-145, Aug. 1982.
- [21] E. Lelarsmee and A. Sangiovanni-Vincentelli, "RELAX: a new circuit simulator for large scale MOS integrated circuits," in *Proc. 1982 Design Automation Conf.*, June 1982.
- [22] G. R. Boyle, "Simulation of integrated injection logic," Univ. of California, Berkeley, ERL Memo. No. ERL-M 78/13, Mar. 1978.
- [23] C. A. Desoer and E. S. Kuh, *Basic Circuit Theory*. New York: McGraw-Hill, 1969.
- [24] K. Sakallah and S. W. Director, "An activity-directed circuit simulation algorithm," in *Proc. IEEE Int. Conf. Circ. Comput.*, pp. 1032-1035, Oct. 1980.
- [25] J. Varga, *Matrix Iterative Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1969.
- [26] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*. New York: Academic Press, 1970.
- [27] A. Householder, *The Theory of Matrices in Numerical Analysis*. Boston, MA: Ginn, 1964.
- [28] A. R. Newton, "The simulation of large-scale integrated circuits," *IEEE Trans. Circuits Syst.*, vol. CAS-26, pp. 741-749, Sept. 1979.
- [29] G. Arnout and H. De Man, "The use of threshold functions and boolean controlled network element for macromodelling of LSI circuits," *IEEE J. Solid-State Circuits*, vol. SC-13, pp. 326-332, June 1978.
- [30] G. Persky, D. N. Deutsch, and D. G. Schweikert, "LTX—A system for the directed automation design of LSI circuits," in *Proc. 13th Design Automation Conf.*, 1976.
- [31] E. Isaacson and H. B. Keller, *Analysis of Numerical Methods*. New York: Wiley, 1966.
- [32] H. Shichman and D. A. Hodges, "Modeling and simulation of insulated gate field-effect transistor switching circuits," *IEEE J. Solid-State Circuits*, vol. SC-3, pp. 285-289, Sept. 1968.
- [33] A. Ruehli, A. Sangiovanni-Vincentelli, and G. Rabbat, "Time analysis of large-scale circuits containing one-way macromodels," *IEEE Trans. Circuits Syst.*, vol. CAS-29, pp. 185-191, Mar. 1982.
- [34] G. De Micheli and A. Sangiovanni-Vincentelli, "Characterization of integration algorithms for the timing analysis of MOS VLSI circuits," *Int. J. Circuit Theory Appl.*, pp. 299-309, Oct. 1982.
- [35] G. De Micheli, A. R. Newton, and A. Sangiovanni-Vincentelli, "Symmetric displacement algorithms for the timing analysis of MOS VLSI circuits," *IEEE Trans. Computer-Aided Design of CAS*, to be published.
- [36] C. W. Gear, *Numerical Initial Value Problems for Ordinary Differential Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1971.
- [37] G. Dahlquist and A. Björck, *Numerical Methods*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
- [38] W. Kahan, private notes, 1975.
- [39] G. De Micheli, "New algorithms for the timing analysis of MOS circuits," M.S. Thesis, Univ. of California, Berkeley, 1980.
- [40] A. R. Newton, "The analysis of floating capacitors for timing simulation," in *Proc. 13th Asilomar Conf. Circuits Syst. Comput.*, Nov. 1979.
- [41] G. De Micheli, A. Sangiovanni-Vincentelli, and A. R. Newton, "New algorithms for timing analysis of large circuits," in *Proc. 1980 Int. Symp. Circuits Syst.*, 1980.
- [42] T. Huang and A. Sangiovanni-Vincentelli, "Analysis of a method for the timing simulation of large-scale MOS circuits containing floating capacitors," in preparation.
- [43] *The CRAY X-MP Series of Computers*, CRAY Research, Inc., Mendota Heights, MN, Pub. MP-0001, 1982.
- [44] *CDC CYBER 200 MODEL 203 Computer System Hardware Reference Manual (Preliminary Edition)*, Control Data Corporation, St. Paul, MN, Pub. 60256010, May, 1980.
- [45] D. A. Calahan and W. G. Ames, "Vector processors: Models and applications," *IEEE Trans. Circuits Syst.*, vol. CAS-26, Sept. 1979.
- [46] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," *Comput.*, vol. 14, Sept. 1981.
- [47] J. B. Dennis, "Data flow supercomputers," *Comput.*, vol. 13, no. 11, pp. 48-56, Nov. 1980.
- [48] I. Watson and J. R. Gurd, "A practical data flow computer," *Comput.*, vol. 15, no. 2, pp. 51-57, Feb. 1982.
- [49] E. Lelarsmee, "The waveform relaxation method for the time-domain analysis of large scale integrated circuits: Theory and applications," Ph.D. dissertation, Univ. of California, Berkeley, 1982; also Memo UCB/ERL M82/40, 1982.
- [50] R. Saleh, "Iterated timing analysis and SPLICE1," M.S. thesis,



- Univ. of California, Berkeley, 1983.
- [52] A. R. Newton, "The simulation of large scale integrated circuits," Ph.D. dissertation, Univ. of California, Berkeley, July 1978; also Univ. of California, Berkeley, Memo UCB/ERL M78/52, July 1978.
  - [53] J. Kleckner and A. R. Newton, "Advanced techniques for iterated timing analysis," in preparation.
  - [54] D. Senderowicz, "An NMOS integrated vector-locked loop," Univ. of California, Berkeley, Memo. No. UCB/ERL M82/83, Nov. 12, 1982.
  - [55] J. White and A. Sangiovanni-Vincentelli, "RELAX2: A new waveform relaxation approach for the analysis of LSI MOS circuits," in *Proc. 1983 Int. Symp. Circuits Syst.*, May 1983.
  - [56] E. Lelarsmee and A. Sangiovanni-Vincentelli, "Some new results on waveform relaxation algorithms for the simulation of integrated circuits," in *Proc. 1982 IEEE Int. Large-Scale Syst. Symp.*, pp. 371-376, Oct. 1982.
  - [57] J. Kaye and A. Sangiovanni-Vincentelli, "Solution of piecewise linear ordinary differential equations using waveform relaxation and Laplace transforms," in *Proc. 1982 Int. Conf. Circuits Comput.*, pp. 180-183, Oct. 1982; also *IEEE Trans. Circuits Syst.*, to be published.
  - [58] M. Guarini and O. A. Palusinski "Integration of partitioned dynamic systems using waveform relaxation and modified functional linearization," *1983 Summer Computer Simulation Proc.*, July 11-13, 1983.
  - [59] T. Y. Feng, "A survey of interconnection networks," *Comput.*, no. 11, pp. 12-27, 1981.
  - [60] D. D. Gajski, D. J. Kuck, and D. A. Padua, "Dependence driven computation," *Proc. IEEE Spring Compcon*, pp. 168-172, Feb. 1981.
  - [61] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 763-776, Sept. 1980.
  - [62] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, no. 12, pp. 1145-1155, Dec. 1975.
  - [63] J. T. Deutsch and A. R. Newton, "Data-flow based behavioral-level simulation and synthesis," *Proc. IEEE ICCAD Conf.*, Sept. 1983.
  - [64] J. T. Deutsch, "Multiprocessor computer architecture," Worcester Polytechnic Institute, MQP Rep. June 1980.

ITERATED TIMING ANALYSIS AND SPLICE1

by

Resve A. Saleh

Memorandum No. UCB/ERL M84/2

4 January 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering  
University of California, Berkeley  
94720

### **Abstract**

SPLICE1 is a mixed-mode simulation program for large-scale integrated circuits. It performs concurrent electrical and logic simulation using event-driven selective-trace techniques. The electrical analysis uses a new algorithm, called Iterated Timing Analysis (ITA), which performs accurate electrical waveform analysis much faster than SPICE2 for large circuits. The logic analysis features a new MOS-oriented state model and a fanout dependent delay model, and handles bidirectional transfer gates in a consistent manner.

This report describes the new algorithms and the details of the implementation in SPLICE1.7. Program performance characteristics and a number of simulation results are also included.

### Acknowledgements

I would like to express my appreciation to my research advisor Prof. A. Richard Newton for his patience, encouragement and guidance throughout course of this work. I would also like to thank Prof. Don O. Pederson and Prof. Alberto Sangiovanni-Vincentelli for their support.

I wish to thank everyone in the CAD group at Berkeley but a few people deserve special mention. In particular, I would like to thank Jim Kleckner for the many long hours of help generating examples, fixing bugs and engaging in useful discussions. I would also like to thank Jacob White for the discussions on the theoretical aspects of the work. I am grateful to Ben Valdez of Hughes Aircraft for reviewing the manuscript carefully, providing useful suggestions for program development and for assistance with examples.

Graeme Boyle, John Crawford, Ian Getreu, Jack Hurt and Steve Potter of Tektronix helped me a great deal during the course of the project. I would also like to express my special thanks to Mike Caughey and the ICCAD group at MITEL Corp. in Ottawa, Canada for their encouragement.

A number of designers have helped during the initial debugging phase of this work. In particular, I would like to thank Don Herbert of Aerospace Corporation, Professor Miles Copeland of Carleton University, Ottawa, Canada, Ron Jerdonek of the General Electric Co., and Marcus Paltridge, Chris Wilson, and Craig Mudge of CSIRO VLSI, Australia.

Both the Digital Equipment Corporation and Toshiba Corporation provided

state-of-the-art test circuits and help with debugging. In particular, I would like to thank Ed Burdick of Digital and Takayasu Sakurai of Toshiba Corporation for their help.

Finally, I wish to thank my wife, Lynn, and my family members in Ottawa, Canada for their continuing support.

This work was supported in part by NSERC (Natural Science and Engineering Research Council) of Canada, the Hewlett-Packard Company, Digital Equipment Corporation and Tektronix.

## TABLE OF CONTENTS

<b>CHAPTER 1: INTRODUCTION .....</b>	<b>1</b>
<b>CHAPTER 2: Iterated Timing Analysis .....</b>	<b>4</b>
2.1 Introduction.....	4
2.2 The Simulation Problem .....	4
2.3 Motivation for a New Simulation Approach.....	5
2.4 Relaxation-based Electrical Simulation.....	14
2.5 The ITA Algorithm.....	15
2.5.1 The Gauss-Seidel Iteration Method .....	15
2.5.2 A Nonlinear Gauss-Seidel Iterative Approach .....	17
2.5.3 The SOR-Newton Iteration.....	21
2.5.4 Convergence of the SOR-Newton Iteration.....	21
2.6 Exploiting Latency.....	23
2.7 Implementation in SPLICE.....	23
2.7.1 Program Flow.....	24
2.7.2 Details of Node Processing.....	24
2.7.3 Element Models.....	25
2.8 ITA Simulation Results.....	26
2.9 Optimizations in the Present Implementation .....	29
<b>CHAPTER 3: Enhancements to the Logic Analysis.....</b>	<b>31</b>
3.1 Introduction.....	31
3.2 The State Model .....	32
3.2.1 A MOS-oriented Logic Model.....	32
3.2.2 State Model Definition.....	34
3.2.3 Using the State Model.....	35

3.3 The Delay Model .....	36
3.3.1 Factors Affecting Switching Delay .....	36
3.3.2 Delay Model for Simple Gates .....	37
3.3.3 Delay Model for Multi-output Elements.....	39
3.3.4 Delay Models for Transfer Gates .....	41
3.3.5 Delay to an Unknown Value.....	48
3.4 Spike Handling.....	48
3.5 Transfer Gate Modeling Issues .....	50
3.5.1 Bidirectional Transfer Gates.....	50
3.5.2 Unknowns at Gate Inputs.....	52
3.5.3 Node Decay .....	54
3.6 Logic Simulation Implementation Details.....	55
3.6.1 General Program Flow .....	55
3.6.2 Node Processing Details.....	56
3.7 Switch-level Simulation .....	57
<b>CHAPTER 4: Examples and Results.....</b>	<b>58</b>
4.1 Program Performance Statistics.....	59
4.2 Profile Statistics .....	61
4.3 Factors Affecting Execution Time in Electrical Simulation .....	62
4.3.1 CPU-time vs. MRT.....	62
4.3.2 CPU-time vs. MINDVSCH.....	63
4.3.3 Effect of Floating Capacitors .....	65
4.3.4 CPU-time vs. SOR.....	67
4.4 SPICE2 vs. SPLICE1.7 .....	67
4.4.1 CDE Circuit .....	68
4.4.2 Digital Filter Circuit.....	72
4.4.3 Industrial Microprocessor Control Circuit.....	74



4.4.4 Industrial 64K CMOS Static RAM Circuit.....	74
4.4.5 NMOS OpAmp Example.....	77
4.4.6 CPU-time vs. Circuit Size.....	83
4.5 Mixed-Mode Examples.....	84
CHAPTER 5: CONCLUSIONS .....	91

## REFERENCES

APPENDIX I: Example Circuits

APPENDIX II: SPLICE1.7 Data Structures

APPENDIX III: SPLICE1.7 Electrical Model Equations

APPENDIX IV: SPLICE1 Source Code

## CHAPTER 1

### 1. INTRODUCTION

SPLICE1 is a *mixed-mode* simulation program for large digital MOS integrated circuits (IC). It performs time-domain transient analysis which tends to be the most time-consuming and memory-intensive task in simulation today. The enhancements made to the program are described in this report. The starting point for this work was SPLICE1.3 [1]. This early version of SPLICE1 included 4-state logic simulation, simple timing analysis and a SPICE-like circuit simulation capability [1, 2].

While this version provided a degree of functionality, it suffered from modeling and accuracy problems intrinsic to the algorithms used in the program. Specifically, the 4-state logic model was not sufficient to perform an accurate true-value logic simulation of general MOS circuits containing transfer gates and wired connections (i.e., more than one gate controlling the state of a node). The simple timing analysis algorithm had inherent accuracy limitations and stability problems and had difficulty analyzing circuits containing floating elements and tight feedback loops. These, and other issues, are examined in detail elsewhere [3], and will be elaborated further in later sections.

The latest version, SPLICE1.7, overcomes these problems by using state-of-the-art algorithms in place of previous ones.

The electrical analysis is performed using a new technique called *Iterated Timing Analysis* (ITA) which can be derived from simple timing analysis [4, 5]. In this approach, the set of nonlinear circuit equations are solved using a relaxation-based method rather than a method which requires the direct solution of a set of linear equations, usually found in standard

circuit simulators such as SPICE2 [6]. ITA is as accurate as SPICE2, assuming identical device models, and has guaranteed convergence and stability properties. Due to the selective trace feature in SPLICE1, the execution time can be up to two orders of magnitude faster than SPICE2, with comparable waveform accuracy, for large circuits. Another key feature of ITA is its ability to perform accurate analysis of complex analog circuits, as will be shown later. Iterated Timing Analysis has shown so much promise that efforts are being directed to generalize it as a standard technique for accurate electrical simulation. Therefore, a matrix-oriented simulation capability is no longer available in SPLICE1.

The logic analysis capabilities have also been extended to include the notion of multiple strengths or impedance levels [3] as is available in most modern MOS-oriented logic simulators [7,8,9,10]. While other simulators usually limit the number of strengths to three, there is practically no limit in SPLICE1.7, which allows up to  $2^{16} - 1$  strengths. More than three strengths are often required to model the interaction between transfer gates of differing geometry [3]. Processing of the gates and nodes proceeds in a manner similar to the electrical analysis. In fact, the logic analysis may be thought of as a relaxation-based method in which the elements are represented by simple logic models rather than complex analytical equations. This concept, together with the idea of multiple impedance levels, allows for a more consistent signal representation and signal conversion in the mixed-mode environment. Clearly, there is a correspondence between an electrical voltage and the logic levels. With the notion of strengths, there is now a natural correspondence between the electrical output conductance of an element and the logic output strength of the element.

SPLICE1 can also be used to perform switch-level simulation [9, 10] to verify circuit functionality at the transistor level. It handles CMOS, NMOS and PMOS circuits in both static and dynamic configurations.

Although SPLICE1 originally included a table look-up scheme to speed up MOS model evaluation [4, 5], it was subsequently dropped from the program. Research on optimal table models and structures is continuing in an independent effort [11] and this feature may be reinstated in a later version. Therefore, this report does not address the issue of table-driven MOS models.

The remainder of this report is divided into four chapters. In Chap. 2, the ITA algorithm is described in detail. The enhancements in the logic analysis are described in Chap. 3. In Chap. 4, a number of simulation results and program performance statistics are presented. Finally, in Chapter 5, the general conclusions are stated with specific mention of future directions.

## CHAPTER 2

### 2. Iterated Timing Analysis

#### 2.1. Introduction

A new form of electrical analysis, called *Iterated Timing Analysis* (ITA), is described in this chapter. The motivation for this work is presented using SPLICE1.3 as an example of Non-iterated Timing Analysis (NTA). A simple mathematical treatment of the ITA method is presented here although a complete mathematical analysis of relaxation-based methods, presented in a rigorous and unified framework, may be found in reference [12]. The details of the implementation in SPLICE1.7 are also included in this chapter.

#### 2.2. The Simulation Problem

The general circuit analysis problem in the time domain requires the solution of a set of first-order nonlinear Ordinary Differential Equations (ODE) of the form:

$$C(v(t), u(t))\dot{v} = -f(v(t), u(t)) \quad (2-1)$$

where

- $v(t)$  is the set of unknown node voltages,
- $u(t)$  is the set of inputs,
- $C(v(t), u(t))$  is the nodal capacitance matrix,
- $f(v(t), u(t))$  is the sum of the currents charging the capacitances at each node.

This formulation can be derived by writing *Kirchoff's Current Law* (KCL) at every node, except the ground node, in a given circuit [12]. The simulation task is to determine the unknown voltages,  $v(t)$ , for every node at every

timepoint due to some input excitation,  $u(t)$ .

The technique used in SPICE2 to solve Eqn. (2-1) is to first convert the set of differential equations into a set of algebraic difference equations using a stiffly-stable integration formula [8]. The nonlinear difference equations are then converted to a set of linear equations of the form:

$$GV = I \quad (2-2)$$

using a damped Newton-Raphson linearization process.  $G$  is the Jacobian matrix (or the small-signal conductance matrix),  $V$  is the unknown voltage vector and  $I$  is the known excitation vector. Next, Eqn. (2-2) is solved using a direct matrix approach to produce the solution vector,  $V$ . Since, in general,  $f(v(t), u(t))$  is a nonlinear function, this process must be repeated until  $V$  converges to a consistent solution.

### 2.3. Motivation for a New Simulation Approach

General-purpose simulation programs, such as SPICE2 [8] and ASTAP [13], have been used extensively to perform accurate circuit analysis for over 10 years. These simulators use direct methods (using sparse matrix techniques) to solve the set of circuit equations. Unfortunately, this approach becomes increasingly expensive as the circuit size increases. The fundamental problem is illustrated in Fig. 2.1. The time required to formulate the set of linear equations grows linearly with circuit size whereas the time required to solve the linear equations is proportional to  $N^k$ , where  $N$  is the number of circuit nodes and  $k$  ranges from 1.1 to 1.5. These two solution phases are referred to in Fig. 2.1 as FORM and SOLVE respectively. The SOLVE phase quickly dominates the total time as the circuit size increases

and this is one reason why the direct approach is not appropriate for large circuits.

Timing simulation was introduced in the mid-seventies to reduce CPU-time at the expense of some accuracy. A new breed of simulators emerged at that time, all tailored to perform transient analysis of large digital circuits [5, 4, 1, 14, 15]. These classical timing analysis programs used iterative techniques [16] to solve the set of circuit equations rather than the direct matrix solution approach of the previous generation. A grounded capacitor was required at every node to guarantee convergence of the method. However, to reduce execution time, none of these programs carried the iteration to convergence and in fact each node equation was solved once at each timepoint.

Large digital circuits typically display a 10-20 % *latency* characteristic. That is, only 10-20 % of the nodes in the circuit are active at any given time. Conceptually, there is temporal sparsity (latency in a waveform over a time period) and spatial sparsity (latency in the network at a given point in time) [17]. Since these iteration methods involve the solution of each equation separately, this latency aspect can be exploited to further improve performance.

Using these techniques, two orders of magnitude of speed improvement was obtained. Accuracy was maintained in these simulators by choosing a small fixed timestep for the entire analysis. This timestep was either constant for all circuits [5] or chosen based on the smallest time constant in the circuit [4]. Later timing simulators adjusted the timestep during the analysis dynamically to limit the voltage change over a timestep to a value specified by the user [1]. Simple timing analysis also relied heavily on the



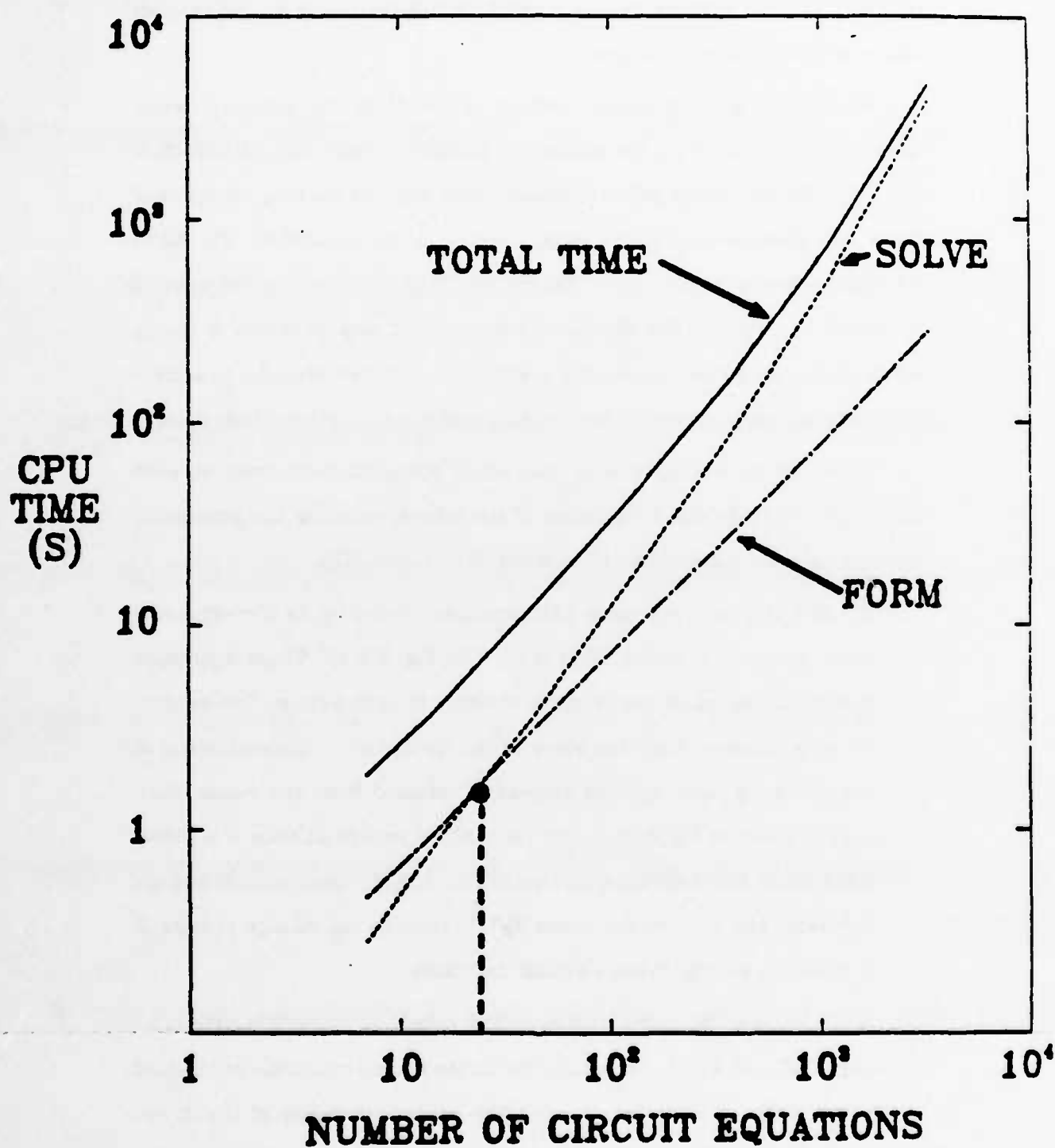


Fig. 2.1 : the amount of CPU time required to perform a transient analysis of a set of typical circuits of increasing size

fact that the accumulated voltage error becomes zero once a node reaches either of the two supply voltages.

While offering a substantial savings in CPU-time and memory usage, these programs suffer from a number of problems which have limited their use. Circuits containing global feedback loops, such as the ring oscillator of Fig 2.2(a), produce timing and voltage errors in the simulation. Fig 2.2(b) illustrates these errors as generated by SPLICE1.3 compared to the solution produced by SPICE2. The SPLICE1.3 program not only produces a timing error (phase error) but incorrectly predicts the number of cycles in a given time period (frequency error) and the height of the peaks (amplitude error).

These errors are all due to the single iteration performed at each timepoint. To understand the origin of the errors, consider the processing sequence of a simple NMOS inverter of Fig. 2.3(a) using NTA.

- (1) The first step is to represent each nonlinear device by its corresponding linear companion model. This is done in Fig. 2.3(b). These equivalent models are based on the terminal voltages of each device. The conductance is obtained from the slope of the nonlinear I-V characteristic at the operating point and the current is obtained from the y-axis intercept as shown in Fig. 2.3(c). The value of the voltage at Node C is calculated using this equivalent circuit of Fig. 2.3(b). Assume that initially  $V_B^{n-1}=0v$  and  $V_C^{n-1}=5.0v$ , where  $V_B^{n-1}$  refers to the voltage at node B at time  $t_{n-1}$  and  $V_C^{n-1}$  has a similar definition.
- (2) Let  $V_B^n=1.0v$ . Then the change in the voltage at node C is calculated using  $V_B^n$  and  $V_C^{n-1}$ . Therefore, the linear equivalent model of the load is the same as it was at  $t_{n-1}$  but the equivalent model of the driver changes. Since the load offers less charging current than it really

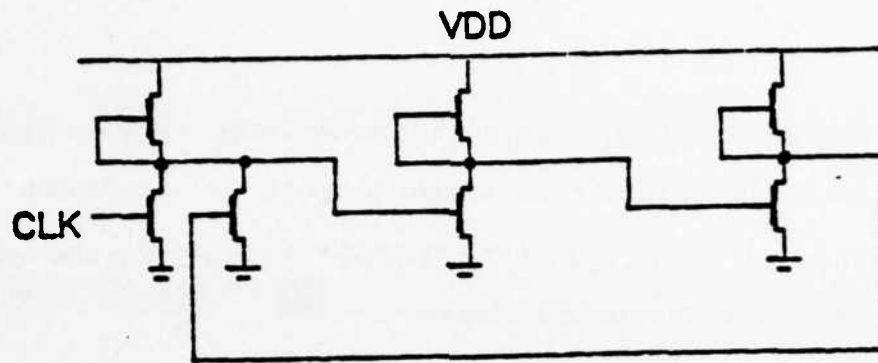


Fig. 2.2(a) : NMOS Ring Oscillator Circuit

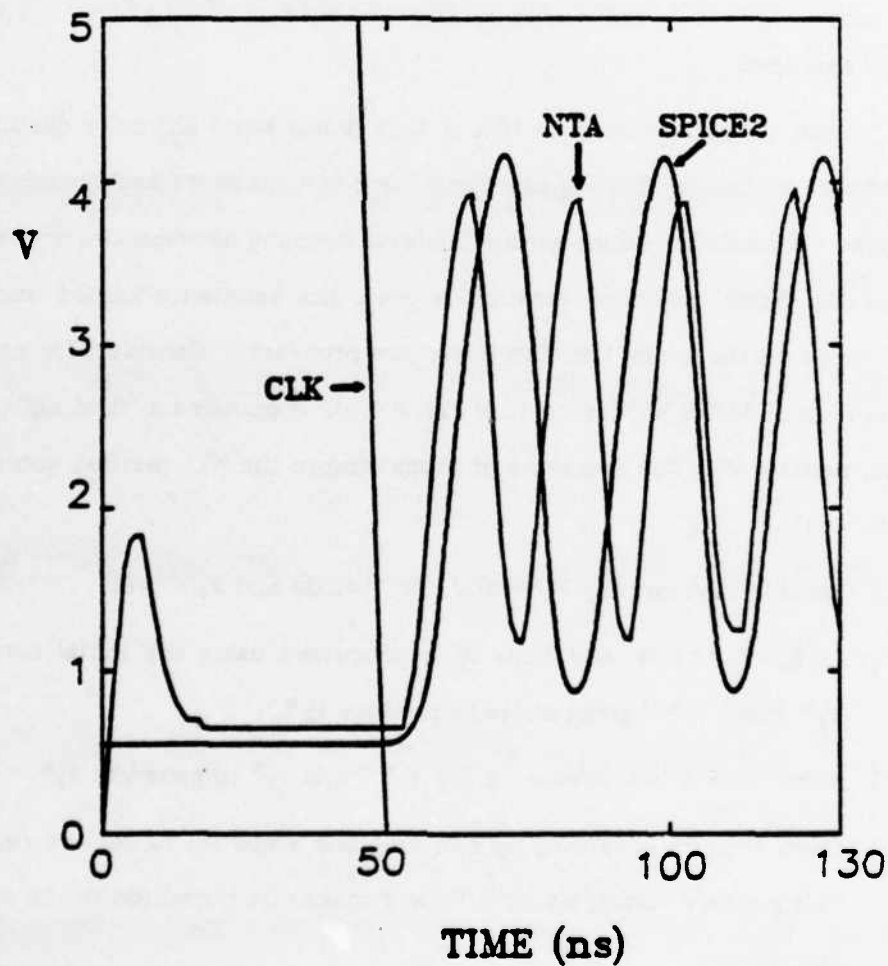


Fig. 2.2(b) : Comparison of NTA and SPICE2. The errors are due to the single iteration performed at each timepoint.

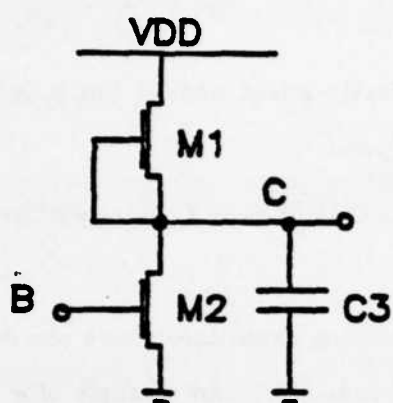
should (i.e.,  $V_C$  is incorrect) and the driver is able to sink more current due to a larger  $V_{ds}$ , the node voltage change  $\Delta V_C^n$  is too optimistic.

- (3) When  $V_B^{n+1}=2.0v$ , again  $V_C^{n+1}=f(V_C^n, V_B^{n+1})$  and  $\Delta V_C^{n+1}$  is also optimistic by the same argument given above.

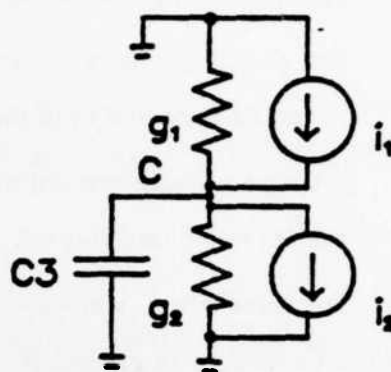
Hence, in a SPLICE1.3 simulation, the output of the inverter will rise and fall earlier in time with faster rise and fall times than the SPICE2 simulation of the same circuit. This error is propagated and intensified in the ring oscillator circuit, resulting in the three errors cited above. It should be noted that if the timestep of the simulation is reduced and the accuracy tolerances are tight, the NTA output will be indistinguishable from SPICE2 output for this example.

Another shortcoming of NTA is that it has some difficulty dealing with circuits containing floating elements, such as capacitors and transfer gates. These elements introduce strong bilateral coupling between two nodes in the circuit. Since only one iteration is used, the solution obtained using NTA depends on the order that the nodes are processed. Consider, for example, the 2-input NMOS NAND circuit of Fig. 2.4(a). It contains a "floating" transistor, namely M2. The sequence of processing in the NTA method would be as follows :

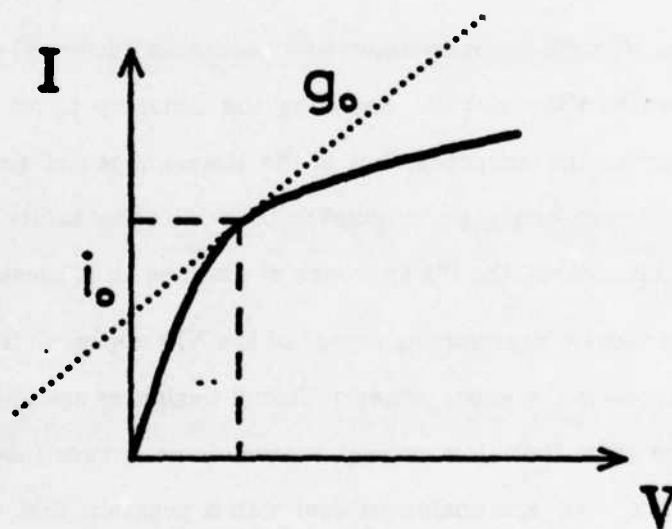
- (1) Assume that initially  $V_X^{n-1}=0v$ ,  $V_Y^{n-1}=5.0v$  and  $V_F^{n-1}=0v$
- (2) At  $t_n$ ,  $V_F^n=1.0v$  and Node X is processed using the initial conditions  $V_X^{n-1}$  and  $V_Y^{n-1}$  given above to produce  $V_X^n$ .
- (3) Next, Node Y is processed using  $V_Y^{n-1}$  and  $V_X^n$  to generate  $V_Y^n$ .
- (4) Then, time is advanced by one unit and steps (2) to (4) are repeated. This process continues until Node Y makes its transition to the opposite rail voltage.



(a)



(b)



(c)

**Fig. 2.3 :** The linear equivalent model for each transistor of (a) is shown in (b) using the device characteristics in (c)

There are two problems with this method:

- the change in node Y should immediately affect node X but it is not reflected at node X until the next time point.
- if the processing started with node Y instead of node X, slightly different results would be obtained.

The same effect is observed when processing capacitors where one node is not connected to ground (i.e., a floating capacitor). An example of a circuit with such a capacitor is the boot-strapped inverter of Fig 2.4(b). The accuracy of NTA depends on the timestep and the ratio of the floating capacitor to the grounded capacitor. In the boot-strapped inverter, the value of C03 is usually large compared to the grounded capacitors, C01 and C02, and this tends to reduce the accuracy of the solution produced by NTA.

Therefore, NTA will produce somewhat inaccurate results when there are floating elements in the circuit. Reducing the timestep to an appropriate value will improve the accuracy, but if the timestep is not small enough, these elements may cause the simulator to exhibit instability. As will be seen later in this section, the ITA approach overcomes all of these problems.

By far the most compromising aspect of the NTA approach is that it may occasionally produce the wrong answer! Circuit designers are willing to use a program which gives them the correct answer or no answer (usually due to non-convergence), but are unable to deal with a program that occasionally produces the wrong answer. In fact, the NTA method *always* produces some answer and this is really the downfall of the method.

For the reasons given above, timing analysis has not been widely accepted as viable form of electrical simulation, although it has been used successfully in constrained IC design methodologies such as standard cell or





gate array. What is really required is a simulation technique which provides both accuracy and speed.

#### 2.4. Relaxation-based Electrical Simulation

A number of new techniques have been developed in an effort to reduce the simulation time while maintaining waveform accuracy comparable to SPICE2. These include table-driven model evaluation [11], microcode tailoring on a minicomputer [18] and the use of vector-oriented computers such as the CRAY-1 [19]. Although these techniques have been successful, they provide, at most, an order of magnitude speed improvement over SPICE2.

Two methods are currently being investigated which use a converged relaxation iteration to solve the set of circuit equations. Both approaches have been implemented and preliminary results indicate that up to two orders of magnitude of speed improvement may be obtained for large digital circuits. One method, called *Waveform Relaxation*[20], decomposes the system of equations into several dynamic subsystems each of which is analyzed for the entire simulation period. The process is then repeated until all the waveforms converge to an exact solution. The relaxation is performed at the differential equation level. This method has been implemented in program RELAX [20,21].

The second method is called *Iterated Timing Analysis* (ITA) [22,23]. In this method, the relaxation is performed at the nonlinear equation level. That is, the set of *nonlinear* circuit equations are iterated to convergence using a Gauss-Seidel or Gauss-Jacobi method. This is also an exact method. Some aspects of this method which make it attractive are as follows:

- it has guaranteed convergence and stability properties
- it allows circuit latency to be exploited easily
- it can be implemented using the concepts developed for logic simulation
- since the logic and electrical analyses operate the same way, a consistent mixed-mode simulation is possible

The algorithm has been implemented in SPLICE1.7 and the implementation details and results obtained are presented in this chapter following a simple mathematical treatment of the method.

## 2.5. The ITA Algorithm

### 2.5.1. The Gauss-Seidel Iteration Method

A system of simultaneous *linear* equations can be solved using a variety of techniques, namely:

1. Direct Methods
  - a. Matrix Inversion
  - b. Gaussian Elimination
  - c. LU decomposition
2. Iterative Methods
  - a. Gauss-Jacobi
  - b. Gauss-Seidel

In circuit simulation, the solution to Eqn. (2-2) is required. The circuit conductance matrix,  $G$ , is usually large but sparse, typically having 3 elements per row. Matrix inversion is not a suitable method because it usually converts a sparse matrix into a dense one. Sparse matrix techniques can be used to solve the equations using method 1(b) or 1(c) but this is not suitable for large circuits due to the rapid increase in CPU-time, as shown in Fig. 2.1.

The iterative methods [16] are well-suited to cases where the matrix is sparse. In fact, the solution of a set of sparse linear equations may be obtained faster using an iterative approach. Two classical iteration methods exist: the Gauss-Jacobi (G-J) method and the Gauss-Seidel (G-S) method. The Gauss-Jacobi method (also referred to in the literature as simultaneous displacement) proposes the following approach:

$$\begin{aligned}
 &v^{(0)} = \text{initial guess voltage vector} \\
 &m \leftarrow 0 \\
 &\text{repeat } \{ \\
 &\quad \text{for } (i = 1 \text{ to } N) \{ \\
 &\qquad v_i^{m+1} = \frac{1}{g_{ii}} \left[ i_i - \sum_{j=1}^n g_{ij} v_j^m \right] \\
 &\quad \} \\
 &\quad m \leftarrow m+1 \\
 &\} \text{ until } |v_i^{m+1} - v_i^m| \leq \epsilon \text{ for all } i, \text{ i.e., convergence}
 \end{aligned} \tag{2-3}$$

Notice that every equation uses the *previous* iteration values for all unknown voltages to obtain a new solution vector. The Gauss-Seidel method (also referred to as successive displacement) suggests the following modification to Gauss-Jacobi:

$$\begin{aligned}
 &v^{(0)} = \text{initial guess voltage vector} \\
 &m \leftarrow 0 \\
 &\text{repeat } \{ \\
 &\quad \text{for } (i = 1 \text{ to } N) \{ \\
 &\qquad v_i^{m+1} = \frac{1}{g_{ii}} \left[ i_i - \sum_{j=1}^{i-1} g_{ij} v_j^{m+1} - \sum_{j=i+1}^n g_{ij} v_j^m \right] \\
 &\quad \} \\
 &\quad m \leftarrow m+1 \\
 &\} \text{ until } |v_i^{m+1} - v_i^m| \leq \epsilon \text{ for all } i, \text{ i.e., convergence}
 \end{aligned} \tag{2-4}$$

Notice that each equation uses the *latest* values of voltage wherever possible.

The only difference in the two methods is whether the previous voltages are always used or the latest values are applied immediately. The convergence rate is linear in both cases but the speed of convergence is quite different. Usually the Gauss-Seidel iteration converges faster than Gauss-Jacobi [16], although there are cases where this is not true.

Both methods also require the strict diagonal dominance condition for guaranteed convergence:

$$\sum_{\substack{j=1 \\ j \neq i}}^n |g_{ij}| < |g_{ii}| \quad (2-5)$$

This inequality states that each diagonal term of the matrix be greater than the sum of all the off-diagonal terms in the same row.

An acceleration scheme is available to speed up convergence using an acceleration parameter,  $\omega$ , as follows.

$$v_i^{m+1} = \omega v_i + (1-\omega) v_i^m \quad (2-6)$$

where  $v_i$  is an intermediate value generated using Eqn. (2-4). The effect of  $\omega$  is usually dramatic but it can only be obtained empirically and usually varies from technology to technology. For standard Gauss-Seidel,  $\omega = 1$ .

### 2.5.2. A Nonlinear Gauss-Seidel Iterative Approach

Relaxation methods, as described in the previous section, can also be applied successfully at the nonlinear equation level. The same approach is used as for linear equations except that each nonlinear equation must first be linearized and solved before proceeding to the next equation. Using this approach, the time-consuming effort required to calculate the Jacobian matrix entries can be avoided.

The steps at the nonlinear equation level are as follows. Starting with equation (2-1), the first step is to convert the differential equations into difference equations using a stiffly-stable integration formula. SPLICE1.7 uses a Backward-Euler formulation [24]. Then the first equation is linearized using the Newton-Raphson (N-R) method and iterated to convergence to solve for one unknown voltage. This constitutes the inner N-R loop. The same process is applied to the next equation and all subsequent equations, in turn, until the last equation is processed. This outer G-S loop is now iterated to convergence to produce the solution.

To further illustrate the method, consider the solution method applied to one node in a typical circuit. Fig 2.5(a) shows three nonlinear devices connected to Node 4, which has a capacitor connected to ground. We begin by writing KCL for Node 4

$$\sum_{j=1}^4 I_j = I_4 + I_1 + I_2 + I_3 = 0 \quad (2-7)$$

This can be rewritten in the form of Eqn. (2-1) :

$$C_4 \dot{V}_4 = -(I_1(V_1, V_4) + I_2(V_2, V_4) + I_3(V_3, V_4)) \quad (2-8)$$

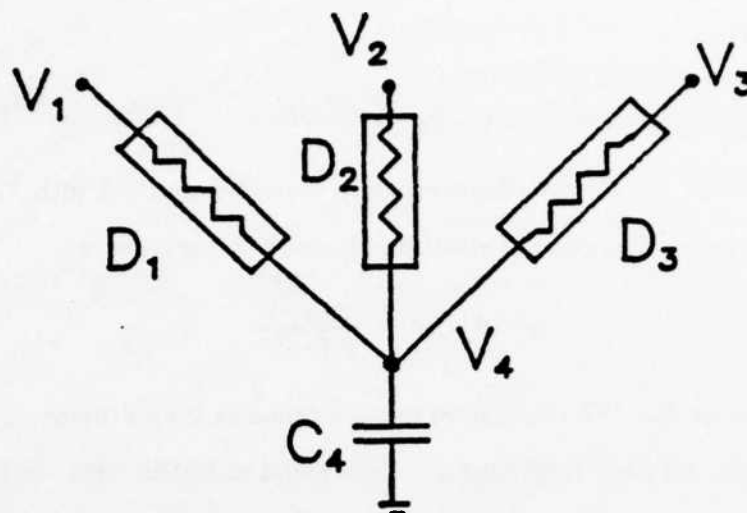
Using the Backward-Euler formula for  $I_4$ , we obtain

$$I_4 = C_4 \frac{dV_4}{dt} = \frac{C_4}{h} (V_{4(n)} - V_{4(n-1)}^*) \quad (2-9)$$

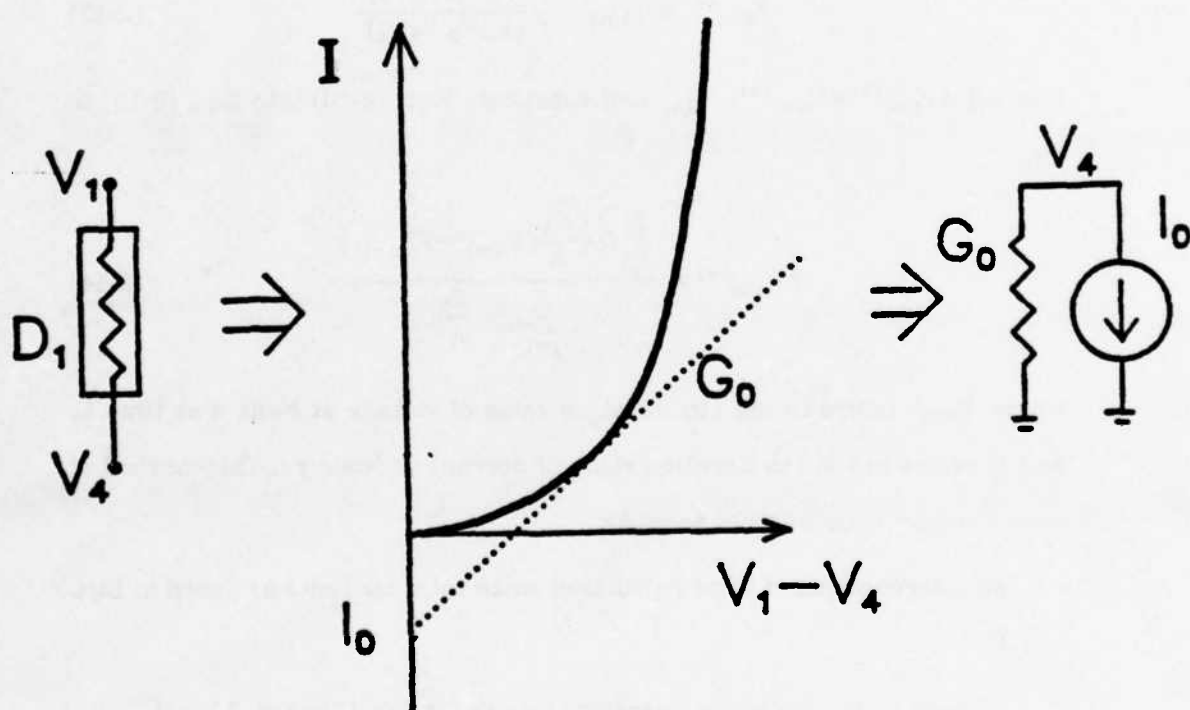
where  $h$  is the integration step size,  $V_{4(n)}$  refers to the voltage value for Node 4 at time  $t_n$  and  $V_{4(n-1)}^*$  refers to the solution obtained for Node 4 at time  $t_{n-1}$ . Therefore, Eqn. (2-8) can now be written as a difference equation,

$$\frac{C_4}{h} (V_{4(n)} - V_{4(n-1)}^*) + I_1(V_1, V_4) + I_2(V_2, V_4) + I_3(V_3, V_4) = 0 \quad (2-10)$$

Since Eqn. (2-10) has the form:



(a)



(b)

**Fig. 2.5 :** The equation used in the Nonlinear Gauss-Seidel iteration is derived using circuit (a). The companion model for each nonlinear device is obtained using the process shown in (b).

$$f(V_1, V_2, V_3, V_4) = 0 \quad (2-11)$$

it is suitable for the Newton-Raphson (N-R) iterative method with  $V_4$  as the unknown variable. The general equation for one N-R iteration is

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})} \quad (2-12)$$

In circuit terms, the N-R calculation usually requires that a linear equivalent be determined for each nonlinear device connected to the node, as shown in Fig. 2.5(b) for D1. This involves the calculation of a conductance,  $G_0$  and a current intercept,  $I_0$ . In order to avoid the intercept calculation, we can apply Eqn. (2-11) directly to Eqn. (2-12) to get

$$V_{4(n)}^{i+1} = V_{4(n)}^i - \frac{f(V_1, V_2, V_3, V_4)}{f'(V_1, V_2, V_3, V_4)} \quad (2-13)$$

Now set  $\Delta V_{4(n)}^{i+1} = V_{4(n)}^{i+1} - V_{4(n)}^i$  and substitute Eqn. (2-10) into Eqn. (2-13) to get

$$\Delta V_{4(n)}^{i+1} = \frac{\sum_{j=1}^3 I_j^i + \frac{C_4}{h}(V_{4(n)}^i - V_{4(n-1)}^i)}{\sum_{j=1}^3 G_j^i + \frac{C_4}{h}} \quad (2-14)$$

where  $V_{4(n)}^i$  refers to the  $i$ th iteration value of voltage at Node 4 at time  $t_n$  and  $I_j^i$  refers to the  $i$ th iteration value of current at Node  $j$ . This method of evaluating  $\Delta V$  is convenient because:

- no intercepts need to be calculated since total currents are used in Eqn. (2-14)
- current levels are within operating ranges (unlike  $I_0$  in Fig. 2.5(b))
- the value of  $\Delta V$  is very accurate when calculated this way. Note that  $\Delta V$  is the difference between two Newton iterations and it will tend toward



zero with each iteration. Therefore it should be calculated as accurately as possible.

For an arbitrary node Eqn. (2-14) becomes

$$\Delta V^{i+1} = \frac{\sum_j B_j + \frac{C}{h}(V_n^i - V_{(n-1)}^i)}{\sum_j G_j + \frac{C}{h}} \quad (2-15)$$

### 2.5.3. The SOR-Newton Iteration

A combination of the Newton-Raphson iteration in a converged Gauss-Seidel loop with acceleration applied is called the SOR-Newton method. In equation form, it is simply

$$\Delta V = -\frac{\omega f(V)}{f'(V)} \quad (2-16)$$

In a standard N-R iteration, the equation is iterated until  $|\Delta V| \leq \epsilon$ . This means that each node equation should be iterated to convergence before moving on to the next one. The Gauss-Seidel loop (i.e., the outer loop) must also be iterated to convergence.

### 2.5.4. Convergence of the SOR-Newton Iteration

A very important property of the SOR-Newton iteration can be applied now to greatly reduce the number of iterations of the inner N-R loop. It happens that *one* Newton iteration per equation for each G-S iteration is sufficient to retain the convergence properties of the nonlinear Gauss-Seidel iteration [16] as long as the convergence requirements of the N-R iteration are strictly satisfied.

A Newton-Raphson iteration will converge if the initial guess is "close enough" to the exact solution, given that the function is Lipschitz

continuous. Under these conditions, the rate of convergence is quadratic. Since the element model equations are smooth, the solution from one timepoint to the next will not be drastically different. Therefore, the solution at the previous timepoint is a good first guess for the N-R iteration. Furthermore, a prediction step may be used to generate a better first guess. A simple linear predictor is used in SPLICE1 using the previous two solution points.

The diagonal dominance requirement for the G-S iteration must also be satisfied to guarantee the convergence of the SOR-Newton iteration. In circuit terms, this requirement can always be met by placing a grounded capacitor at every node and choosing an appropriate timestep. Grounded capacitors appear as  $\frac{C}{h}$  terms in the diagonal position of the conductance matrix  $G$ . Therefore,  $h$ , which is the simulation timestep, can be reduced until the  $\frac{C}{h}$  term is greater than the sum of all off-diagonal terms.

Off-diagonal terms appear in the conductance matrix when there is coupling between two nodes. For example, when floating capacitors are used,  $\frac{C}{h}$  terms appear in diagonal and off-diagonal positions. Therefore, reducing the value of  $h$  is not as effective and this may lead to convergence problems. The ratio of the floating capacitor to the grounded capacitor is an important factor in determining the speed at which convergence is achieved. If the floating capacitor is very large compared to the grounded capacitor, convergence speed will be slow, if the iteration converges at all. The current version of SPLICE uses the IIE method (Implicit-Implicit-Explicit) [25] to evaluate floating capacitors.

## 2.6. Exploiting Latency

SPLICE1 does not solve every node at every timepoint. In fact, only those nodes which are active at any given point in time are processed. Since large circuits are relatively inactive, less than 20% of the nodes are actually solved at each timepoint. The active nodes are determined on an event-driven basis. That is, a node is placed in the set of active nodes if any node which can affect it changes by a significant amount.

Once the set of active nodes are identified, SPLICE1 can exploit two forms of latency. The first one is called simply *latency in time*. This is based on the fact that digital circuit waveforms feature long constant periods. An active node is processed at consecutive points in time until it reaches a constant value. It is then removed from the set of active nodes and becomes latent. The second form of latency is the so-called *latency at a timepoint*. This refers to the fact that some nodes may actually converge with fewer iterations than others, at a given timepoint. These nodes can be marked to be processed at the next timepoint while the remaining nodes continue to iterate to convergence at the current timepoint. Tightly-coupled nodes usually require more iterations than other nodes.

The decoupled nature of ITA allows both forms of latency to be exploited efficiently. These techniques reduce the overall computation significantly. Of course standard circuit simulators solve every node at every timepoint and all nodes converge simultaneously.

## 2.7. Implementation in SPLICE

The analysis techniques described in the previous sections have been implemented in SPLICE1.7. The details are described in this section with

special attention given to areas where further optimization would improve the simulator performance.

SPLICE1 has a fixed minimum timestep called the **mrt** (minimum resolvable time). Events can only be scheduled at integer multiples of **mrt**. There is a scheduling threshold parameter called **mindvsch** which is the minimum change in a node voltage over a timestep which causes the fanout elements of the node to be scheduled. The convergence criterion is defined by two parameters called **abstol** (absolute tolerance) and **reltol** (relative tolerance).

### 2.7.1. Program Flow

The program flow has not changed since the SPLICE1.3 release. The details of the processing may be found in [1,3] and are not repeated here. The data structures of the ITA as implemented in SPLICE1.7 are given in APPENDIX II. The general program flow for electrical analysis is as follows:

```

set all nodes to their initial values ;
schedule all FOL's at time 0 ; #FOL = FanOut List of a node
 $t_n \leftarrow 0$  ;
while (  $t_n < \text{TSTOP}$  ) {
  foreach (FOL in the queue at the current timepoint) {
    foreach (element in the FOL) {
      foreach (output node of an element) {
        process node ; #see next section for details
        schedule FOL if necessary ;
      }
    }
  }
  plot all requested active nodes ;
   $t_n \leftarrow t_n + 1$  ;
}

```

### 2.7.2. Details of Node Processing

A subroutine in SPLICE1.7 processes all electrical nodes, calculates the new node voltage, decides whether the node has converged and determines

whether subsequent scheduling is necessary. A high-level pseudo-code description of the routine is as follows:

```

begin
# Iterated timing analysis algorithm in SPLICE1.7
# Node processing sequence
  obtain next node  $m$ ;
  if (first time processed at new timepoint) {
    use last two points to perform linear prediction;
    convlg=false;
  }
  Gnet = Inet = 0;
  for ( each fanin element at node  $m$  ) {
    compute equivalent conductance  $G_{eq}$ ;
    compute total current flowing into node  $l_{eq}$ ;
    Gnet = Gnet +  $G_{eq}$ ;
    Inet = Inet +  $l_{eq}$ ;
  }
  calculate  $\Delta V$ ; #change in voltage over an iteration
   $V_n^{(i+1)} = V_n^{(i)} + \Delta V$ ; #new node voltage
   $DV = |V_n^{(i+1)} - V_{n-1}|$ ; # change in node voltage over one timestep
  if ( $\Delta V < \text{tolerance}$ ) { # node has converged
    if (convlg = false) { #have not converged at this timepoint before
      if ( $DV > \text{mindvsched}$ ) { # node change is significant
        schedule current fol at  $T_{n+1}$  (future);
        schedule fol of node at  $T_n$  (now);
        convlg = true;
      }
      else { # node change is not significant over one timestep
        do nothing;
      }
    }
    else #have converged previously at this timepoint
      do nothing; #break any feedback loops
  }
  else { # node has not converged so keep processing
    convlg = false;
    schedule current fol at  $T_n$  (now);
    schedule fol of node at  $T_n$  (now);
  }
}
# Finished this node for this iteration
return
end

```

### 2.7.3. Element Models

SPLICE1.7 has built-in models for resistors, linear capacitors (floating and grounded), diodes and MOS transistors. The IIE method is used for

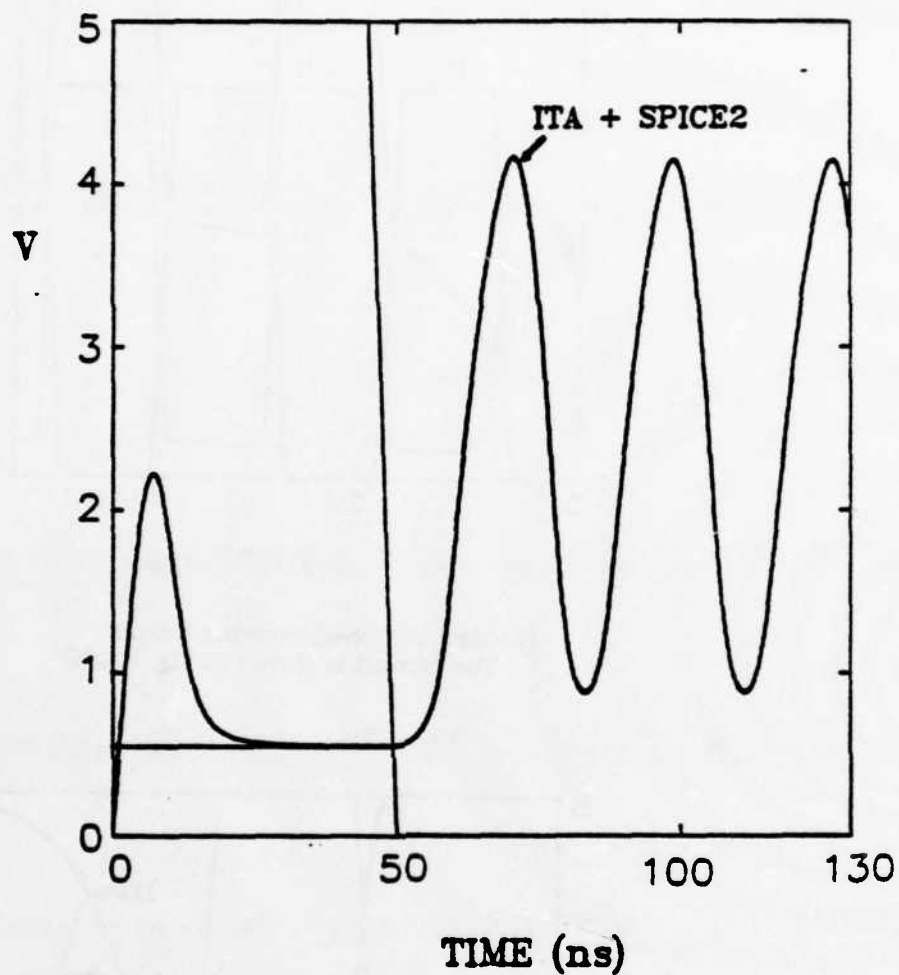
floating capacitors [25] and the first-order Schichman-Hodges model [26] equations are used to model MOS transistors. The model equations for each device are given in APPENDIX III.

Each electrical element has a corresponding program subroutine. The subroutine evaluates the linear equivalent model for each nonlinear device and returns it to the calling routine. As mentioned previously, the intercept current calculation can be avoided by a simple reformulation of the equations. Using this approach, the conductance and the *total* current at a given operating point is returned by each subroutine. The calculation of the equivalent model assumes that all other nodes have ideal *constant* voltage sources attached to them, except in the case of floating capacitors, since IIE is used.

## 2.8. ITA Simulation Results

This chapter has been concerned mainly with simulation accuracy and would not be complete without a comparison of ITA with SPICE2. Fig. 2.6 shows the simulation results obtained for the ring oscillator, 2-input NAND and boot-strapped inverter circuits described earlier. As indicated by the results, SPLICE1.7 produces results which are indistinguishable from those obtained by SPICE2 except at timepoints near time zero due to different initial value assumptions. Therefore, circuits which handled inadequately using NTA do not pose a problem to ITA in terms of accuracy.

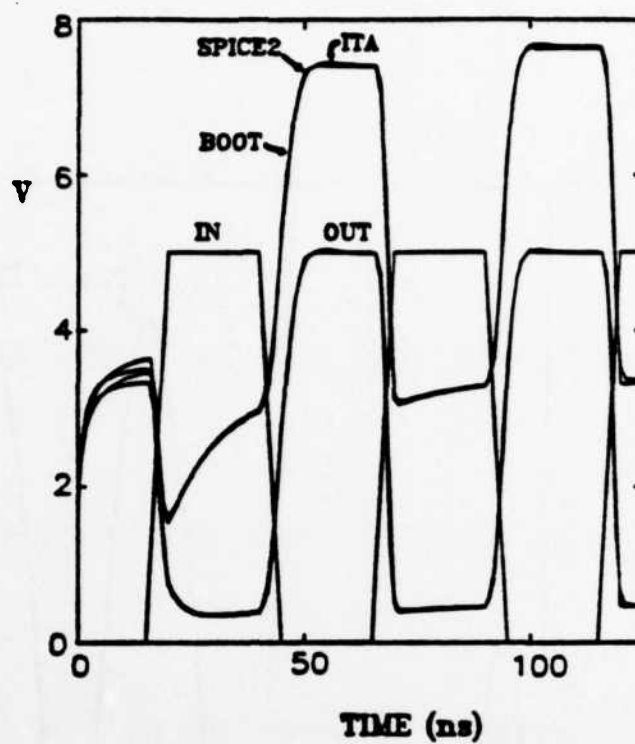
The run-times of the 3 examples do not demonstrate the speed advantage of ITA because the circuits are all very small with dense G matrices and small circuits tend to be very active. The selective trace feature in SPLICE is a significant advantage in very large circuits.



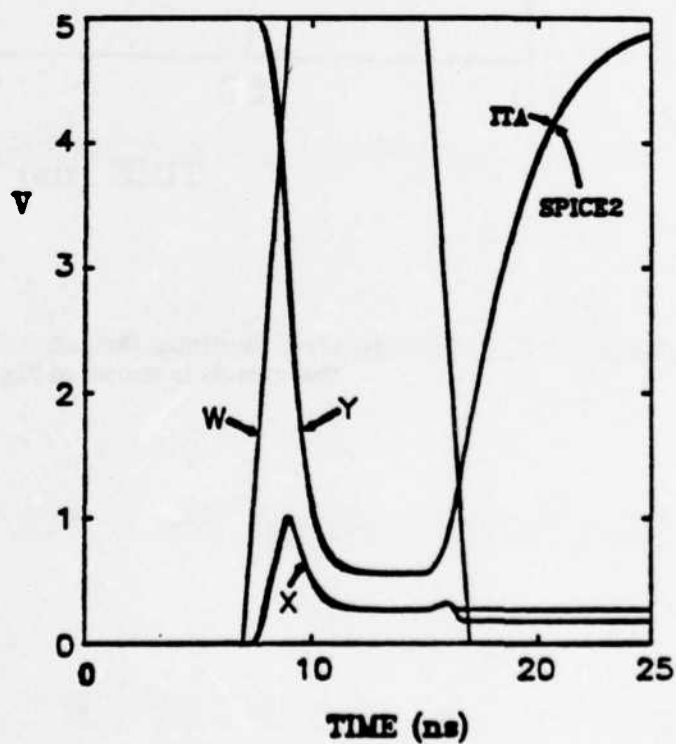
(a) Ring Oscillator Output.  
The circuit is shown in Fig. 2.2(a)

Fig. 2.8: A comparison of the accuracy of ITA vs. SPICE2.  
NTA had problems with each circuit.





(b) Boot-strapped Inverter Output.  
The circuit is shown in Fig. 2.4(b)



(c) NAND circuit Output.  
The circuit is shown in Fig. 2.4(a)

## 2.9. Optimizations in the Present Implementation

While the data structures used in SPLICE1 are well-suited to handle circuit, timing and logic simulation concurrently, they are not ideal for ITA. If a separate program were written to perform ITA, several optimizations could be made to improve the program performance.

For example, some nodes may be reprocessed after they have converged because there may be several paths to the same node through different elements. A node may also be processed many times in succession before another node is processed (i.e., two or more Newton iterations). Furthermore, there are many levels of indirection which must be traversed in order to reach a node, as shown in a previous section.

These problems can be eliminated by scheduling and processing *nodes* as opposed to fanout lists. One such scheme which uses two buffers,  $E_A$  and  $E_B$ , avoids reprocessing a node before all other active nodes are processed:

```

put all nodes in event list  $E_A(0)$ ;
 $t_n \leftarrow 0$ ;
while (  $t_n < TSTOP$  ) {
     $k \leftarrow 0$ ;
    while ( event list  $E_A(t_n)$  is not empty ) {
        foreach (  $i$  in  $E_A(t_n)$  ) {
            obtain  $\Delta V$ ;
             $v_i^{k+1} = v_i^k + \Delta V$ ;
            if (  $|v_i^{k+1} - v_i^k| \leq \epsilon$  ) { i.e., if convergence is achieved
                add node  $i$  to list  $E_A(t_{n+1})$ ;
            }
            else {
                add node  $i$  to event list  $E_A(t_n)$ ;
                add fanout nodes of node  $i$  to event list  $E_A(t_n)$ 
                if they are not already there;
            }
        }
         $E_A(t_n) \leftarrow E_B(t_n)$ ;
         $E_B(t_n) \leftarrow \text{empty}$ ;
    }
     $t_n \leftarrow t_{n+1}$ ;  $t_{n+1} = \text{next timepoint}$ 
}

```

Another shortcoming of the current implementation is that if a node does not converge at a timepoint, the program simply stops execution. The user must decrease the timestep manually and re-run the entire simulation. An automatic internal timestep control mechanism would be useful not only for the convergence problem but also for error control. If the error is small at a particular timepoint, then the timestep could be increased. If the error is too large, the timestep could be decreased. The nodes would then be re-evaluated at the new timepoint. Hence, the timestep could be computed based on an estimate of the Local Truncation Error. In fact, each node could have its own mrt, independent of other nodes, as long as some consistency is maintained in the simulation between different nodes. Unfortunately, dynamic timestep control requires the ability to "backup" in time (i.e., a buffering of previous results for each node) and requires a modification of the data structures to allow successive refinement of the mrt (minimum resolvable time) in the time queue [27]. For this reason, it would require a considerable amount of effort to test this scheme in the current SPLICE1 environment.

## CHAPTER 3

### 3. Enhancements to the Logic Analysis

#### 3.1. Introduction

The improvements in the logic analysis of SPLICE1 are described in this chapter. The starting point for this work was SPLICE1.3. It had the following features:

- a 4-state logic model (0,1,X,Z)
- fixed assignable rise and fall delays on all gates
- unidirectional and some bidirectional elements handled.

There have been a number of changes in the logic analysis since the SPLICE1.3 release. These changes were made to alleviate some of the problems in the previous version and to facilitate conversions in the mixed-mode environment.

The new version is SPLICE1.7 which features:

- a new MOS-oriented state model
- a fanout dependent delay model
- unidirectional and generalized bidirectional element processing.

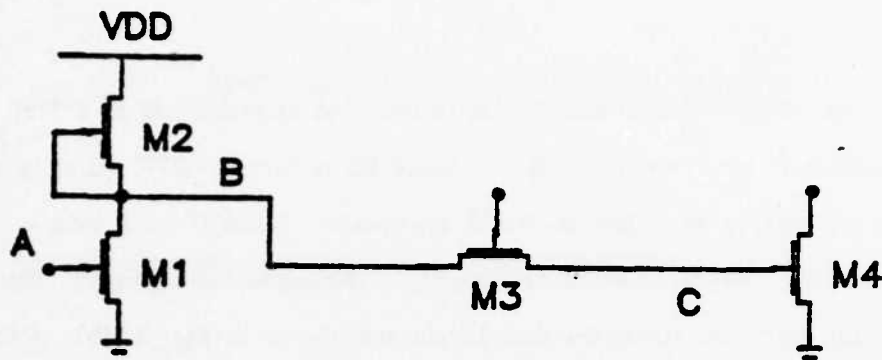
The logic analysis is performed using a relaxation-based method, similar in nature to the electrical analysis. In fact, the logic analysis can be thought of as an implementation of non-iterated timing analysis (see Chap. 2) with simplified element models. Each logic node carries information about the node voltage and the equivalent conductance-to-ground, as does the electrical node. Therefore, the mixed-mode interface is defined in a consistent manner in SPLICE1.7.

This chapter begins with a description and definition of the new state model. Following this, the delay model is described. Next, the "spike" detection and handling procedure is presented. A spike is a pulse at a node of shorter duration than the minimum width necessary to trigger subsequent gates. This is usually an error condition which must be identified and reported to the user. In the next section, the important issues pertaining to the MOS transfer gate are reviewed. The transfer gate (or transmission gate, or pass transistor) is the source of many MOS modeling problems at the logic level and the reason for this will become clear in this chapter. The logic analysis algorithm will then be presented in the section which follows. SPLICE1.7 can also be used to perform switch-level simulation [9, 10] and this is described in the last section. Background material on MOS logic simulation may be found in reference [3].

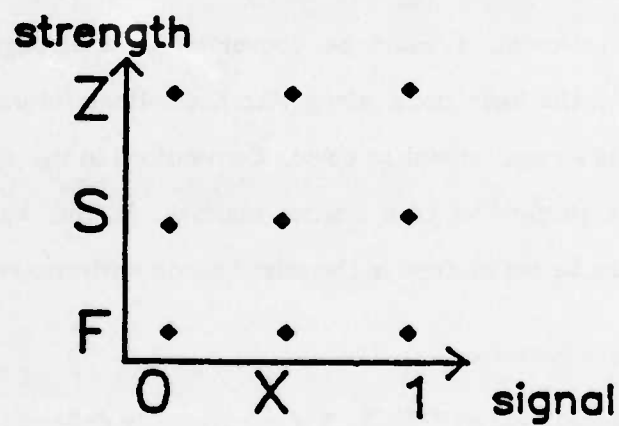
### 3.2. The State Model

#### 3.2.1. A MOS-oriented Logic Model

Most modern logic simulators handle the problems specific to MOS integrated circuits by including the notion of *signal strength* [7, 8, 9, 10] in the logic model. The rationale for this has been presented in a previous publication [3]. Strength is an abstraction of the large-signal conductance from a node to ground or from a node to a supply voltage. It can be associated with the output of a gate or it can be an attribute of a node. For example, in the inverter of Fig. 3.1 (M1 and M2), the driver transistor with its gate input at 5.0V represents a very low resistance path from Node B to ground. In MOS logic model terms, this is referred to as a "forcing 0" or "driving 0". Similarly, the load transistor represents a sizeable resistance from Node B to VDD



(a)



(b)

**Fig. 3.1 :** The circuit in (a) illustrates the use of the strength-oriented MOS model. The graph in (b) shows the relationship between the strengths and levels in a 9-state logic model.

(approx.  $20k\Omega$  to  $40k\Omega$ ) and this is referred to alternatively as a "soft 1", a "resistive 1" or a "weak 1". If transistor M3 is turned "OFF" (that is, if the gate voltage is zero for an NMOS transistor), Node C goes into a "high-impedance" condition which represents a third distinct strength. The relationship between strengths and levels are shown in Fig. 3.1(b). Although most simulators are based on these three strengths, SPLICE1.7 allows up to  $2^{16}-1$  strengths for two reasons:

- there is a requirement for more than three strengths when modeling the interaction of several transfer gates with differing W/L ratios, typically found in bus contention situations.
- it provides a mechanism for consistent signal representation in the logic domain for schematic or mixed-mode simulation [22]. If information about the effective conductance to ground is stored with each electrical node, this information could be converted to a strength value and passed on to the logic node, along with the voltage information, whenever there is a requirement to do so. Conversions in the opposite direction can be performed in a similar manner. In this way, simulation accuracy can be maintained in the mixed-mode environment.

### 3.2.2. State Model Definition

The state model used in SPLICE1.7 is now formally defined :

- A state is composed of a logic level, logic strength pair (L,S).  
i.e.,  $state=(L,S)=(Level, Strength)$
- The logic level can be one of three values: logic zero(0), logic one(1) or logic unknown(X). The "0" level represents the low threshold value or ground. The "1" level represents the high threshold value or VDD. The



"X" level represents an undetermined value which could be "0", "1" or some value in between. The logic level field is extracted from the state using the "lev" function. That is,

$$L = \text{lev}(\text{state})$$

- The logic strength is an integer value between 1 and some user-specified upper limit. The upper limit has a maximum allowed value of 65,536. In this report, the subscripts  $F$ ,  $M$  and  $H$  will be used to denote the largest, middle and smallest strengths respectively in a given range. The strength field is extracted from the state using the "str" function. That is,

$$S = \text{str}(\text{state})$$

- An initial unknown,  $X_i$ , must be distinguished from an unknown generated during the analysis,  $X_g$ . This is done in SPLICE1.7 by defining the initial unknown as follows :

$$X = \text{lev}(\text{initial\_unknown})$$

$$0 = \text{str}(\text{initial\_unknown})$$

and the generated unknown as follows:

$$X = \text{lev}(\text{generated\_unknown})$$

$$0 \neq \text{str}(\text{generated\_unknown})$$

The initial unknown is useful to identify nodes which are not exercised by the input pattern used in a simulation. As a post-processing step, these nodes could be reported to the user.

### 3.2.3. Using the State Model

In a logic analysis, nodes are scheduled to be processed in the time queue in accordance with the activity in the circuit. When a node is

processed, the fanin list (FIL) is obtained from the node data structure (see Appendix II, parts 1,2). Each gate in the fanin list is a potential "driver" of the node (definition of a fanin) but usually only one gate will gain control of the node and determine its final state. The gate with the largest output strength is declared the "winner" and the node adopts the output state of the winning gate. Node contention occurs when two or more gates attempt to drive the same node to different logic levels with the same driving strength. In this case, the node is assigned an X level and the strength of any one of the contending gates. The processing details are presented in Section 3.6.

### **3.3. The Delay Model**

#### **3.3.1. Factors Affecting Switching Delay**

Once a new state is determined, the next task is to calculate the time required to reach the new state. In MOS circuits, the switching time is based on many factors which include:

- the basic gate switching time (unloaded)
- the static output loading due to capacitance of elements connected to the node
- the dynamic output loading through transfer gates which are turned "ON" (that is, transistors with their gates at the logic level "1")
- the number of gate inputs
- the shape (rise and fall times) of input waveforms

No logic simulator attempts to incorporate all of the above factors into the delay calculation. On the other hand, it is essential that a logic simulator

include all the first-order effects in the delay calculation. SPLICE1.7 is capable of modeling the effects due to the first four factors. The fifth factor (input waveform shape) is more difficult to handle at the logic level, although it may be a significant factor in many cases.

### 3.3.2. Delay Model for Simple Gates

The usual modeling procedure for logic simulation is to generate a set of curves similar to Fig. 3.2 for every primitive element (NANDs, NORs, inverters, etc.) using accurate electrical simulation. In this figure, the delay from the input switching point to the output switching point is plotted as a function of output loading and the number of inputs. A step voltage is assumed as the input of the gate. Although not strictly true, the relationships are usually taken to be linear. The y-intercept of each curve represents the intrinsic unloaded gate delay while the slope of each curve represents the gate drive-capability.

Assuming that the above information is available, the following method can be used to calculate delays for simple gates. The first requirement is that a capacitance value be specified on every input and output pin of every gate as part of the model definition. Then the total gate delay can be represented by four parameters : the intrinsic gate delays ( $t_r$ ,  $t_f$ ) and the gate drive-capabilities ( $t_{rc}$ ,  $t_{fc}$ ),

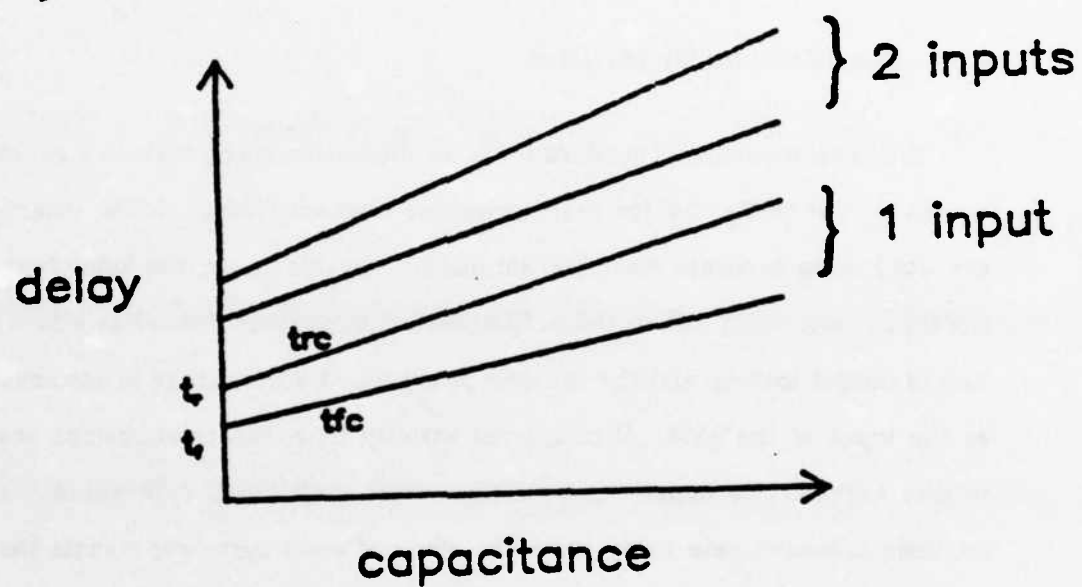
where

$t_r$  = rise time for unloaded gate (intercept)

$t_f$  = fall time for unloaded gate (intercept)

$t_{rc}$  = gate drive-capability for rising signals (slope)

$t_{fc}$  = gate drive-capability for falling signals (slope)



**Fig. 3.2: Typical delay curves generated for a logic gate using electrical analysis**

Using these values, the total delay is calculated using the equation:

$$rissetime = tr + trc * (node \ capacitance) \quad (3-1a)$$

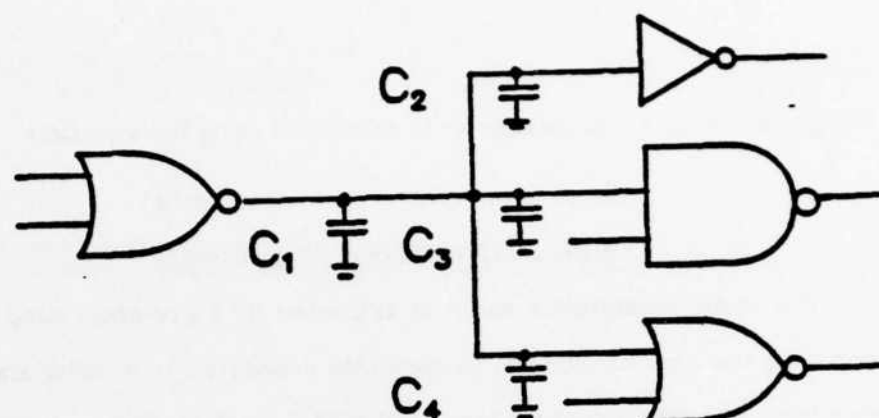
$$falltime = tf + tfc * (node \ capacitance) \quad (3-1b)$$

The node capacitance value is extracted in a pre-processing step by summing the capacitances of all elements connected to a node, and stored with the node data structure (see APPENDIX II, part 1). This process is illustrated in Fig 3.3.

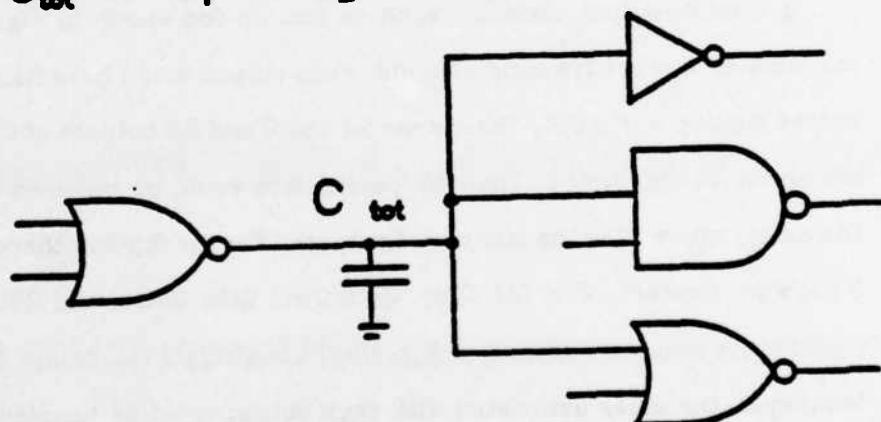
### 3.3.3. Delay Model for Multi-output Elements

If a multi-output element, such as the flip-flop shown in Fig. 3.4(a), is available as a primitive logic element, each output would have its own set of curves similar to Fig 3.2. The curves for the Q and QB outputs of the flip-flop are shown in Fig. 3.4(b). Then  $4N$  parameters would be required to specify the delay, where  $N$  is the number of outputs. For the flip-flop there would be 8 such parameters : Qtr, Qtf, QBtr, QBtf, Qtrc, Qtfc, QBtrc, and QBtfc. These parameters would be applied to Eqn. (3-1) to calculate the delay. Using this technique, the delay associated with each output could be handled independently. The overriding assumption is that the rise and fall drive-capabilities (trc,tfc) of the outputs are constant and independent of the inputs.

In certain elements, the delay from a particular input (say, the RESET pin of the flip-flop) to a given output (either Q or QB) is different from another input to output delay (J- or K-input to Q delay). This suggests that, in fact, the intrinsic delay should be a matrix which is indexed by input pin which initiates activity and the output pin being processed. Then the total delay due to loading could be calculated using Eqn. (3-1) and the specific trc and tfc values for each output. This is shown in Table 3.1 below for the flip-



$$C_{\text{tot}} = C_1 + C_2 + C_3 + C_4$$



**Fig. 3.3 :** SPLICE1 preprocesses the input and output capacitances specified for each gate and uses the total capacitance at the node to calculate fanout dependent delays.

flop example.

Table 3.1 Intrinsic Delay Matrix

I/O pin	J	K	CLK	RESET	SET
Q	tr=10 tf=10	tr=10 tf=10	tr=10 tf=10	tr=5 tf=6	tr=5 tf=6
QB	tr=10 tf=11	tr=10 tf=11	tr=10 tf=11	tr=5 tf=6	tr=5 tf=8

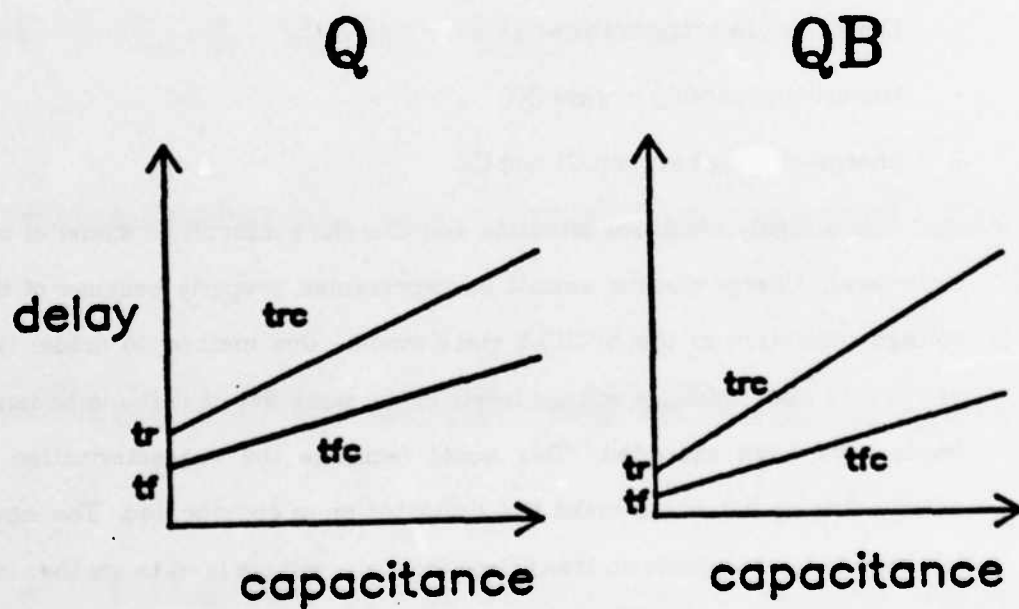
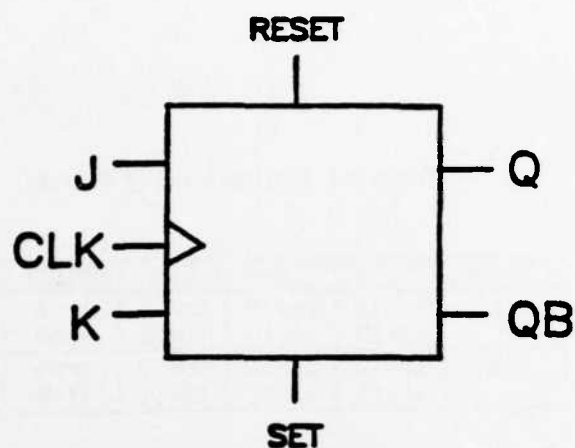
### 3.3.4. Delay Models for Transfer Gates

The delay calculation for logic circuits containing transfer gates is more complex than either of the two cases given above. Consider the circuit of Fig. 3.5. The delay from Node A to Node B when the input CLK of the transfer gate makes a transition from "0" to "1" is based on:

- the W/L ratio of the transfer gate
- the drive-capability of gate INV
- charge-sharing between C1 and C2

It is a highly nonlinear situation and therefore difficult to model at the logic level. Charge-sharing cannot be represented properly because of the voltage resolution in the SPLICE1 state model. One method to model this effect is to allow multiple voltage levels in the same way that the impedance levels have been extended. This would facilitate the characterization of charge-sharing but would make the simulator more complicated. The simulator would have to perform transitions from one voltage level to another in a consistent manner. SPLICE1.7 lumps all the nonlinear effects into two values called the turn-on ( $t_{on}$ ) and turn-off ( $t_{off}$ ) times. These values do not take capacitive effects into account.





**Fig. 3.4 :** For the JK Flip-Flop of (a), a set of delay parameters are required for each output, Q and QB, as shown in (b).

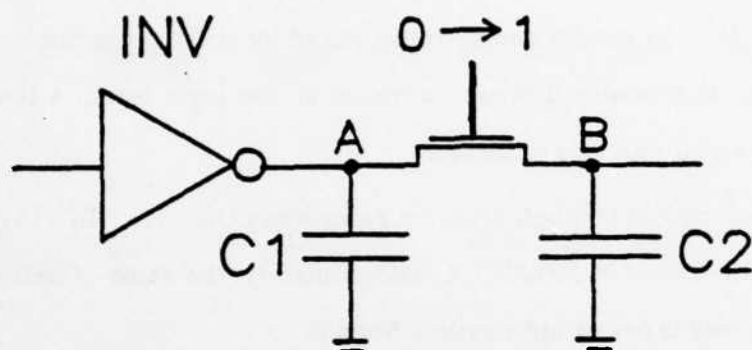
Another delay modeling issue concerns transfer gates connected in series as shown in Fig. 3.6. The delay in question is that from Node A to Node E. If all gates are "ON", the circuit can be represented by an RC transmission line. Unfortunately, this is also difficult to model at the logic level. A few alternatives exist to deal with this situation:

- Use a zero delay model through transfer gates when they are "ON" [8]. This is the method used in SPLICE1.7. Unfortunately, the value of delay calculated this way is overly optimistic at Node E.
- Lump capacitances C1, C2, C3, C4 and C5 together and use this value in Eq. (3-1). This is the transition delay for all nodes from the old state to the new one. The value of delay calculated this way is overly pessimistic at Node A.
- Extend the notion of drive-capability of a gate to nodes other than its output node. Since tx1 is "ON", both Node A and Node B are driven by gate INV. Therefore, the delay to A could be calculated as given in eq. (3-1) and the delay to B could be calculated using the equation:

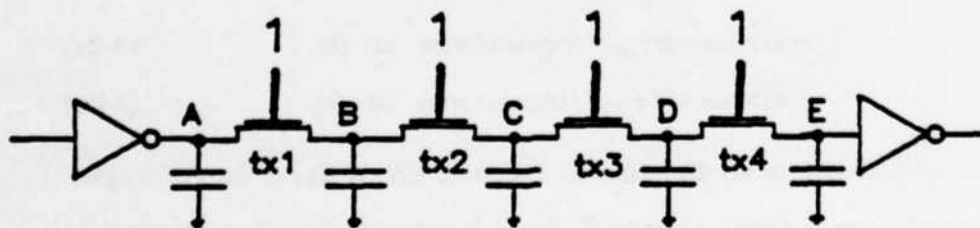
$$risetime = trc_{INV} * (capacitance \text{ at } B) \quad (3-2a)$$

$$falltime = tfc_{INV} * (capacitance \text{ at } B) \quad (3-2b)$$

To compute the delay to nodes C, D and E, simply apply Eq. (3-2) again using the capacitance at node C, D and E respectively. This approach is better than either of the above methods but is still lacking in accuracy because it does not account for the "ON" resistance of the transistors. One modification which may provide more accuracy is to adjust the values of  $trc$  and  $tfc$  using the "ON" resistance of the transfer gates and the depth of the node away from the output of the controlling node. This method is promising because VLSI circuits typically contain



**Fig. 3.5 :** The delay from Node A to Node B, when the transfer gate turns on, is due to highly nonlinear effects which are difficult to model at the logic level.



**Fig. 3.6 :** Delay associated with series-connected MOS transfer gates is difficult to model due to the RC transmission line characteristics.

interconnections which are electrically equivalent to distributed RC transmission lines. This interconnect delay dominates the total delay for very large circuits. It could be modeled the same way as the set of series transfer gates. Therefore, a netlist extractor could provide the simulator with "DELAY" elements, as shown in Fig 3.7, in place of interconnect with delay calculations performed using the modified eq. (3-2) :

$$risetime = TRC * (capacitance \text{ at node}) \quad (3-3a)$$

$$falltime = TFC * (capacitance \text{ at node}) \quad (3-3b)$$

where

$$TRC = trc * f(resistance, depth)$$

$$TFC = tfc * f(resistance, depth)$$

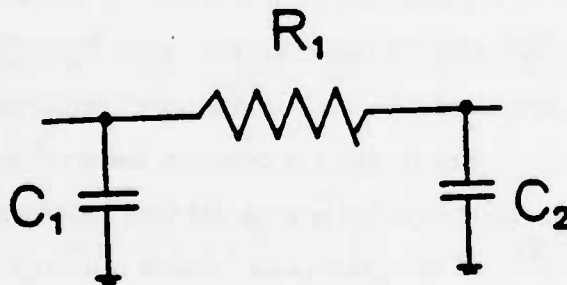


Fig. 3.7 : A proposed equivalent model for a delay element.

### 3.3.5. Delay to an Unknown Value

The delay calculations in the previous section assume signal transitions from "0" to "1" or "1" to "0". Nodes may, of course, acquire the X level due to contention at the node as described earlier. The question then arises as to when the X level takes effect. The unknown level could be "0", "1" or some intermediate value. Clearly, if the unknown is the previous value, there is no delay. If it is a new logic level there is a rise or fall transition delay. The usual approach is to assume that the unknown value takes affect immediately (as is done in SPLICE1.7) or one time unit in the future.

### 3.4. Spike Handling

SPLICE1.7 uses an inertial delay algorithm. This means that if a node is scheduled to change at some time in the future  $T_n$ , it is held at its old value until that time. Then at  $T_n$ , the new value is assigned to the node and the fanouts of the node are processed using this value. A *spike* (commonly referred to as a glitch) occurs if the node is scheduled to change to a different value before it reaches the new value. Spike detection is simple in true-value logic simulation but becomes very complicated when performing fault simulation. When a spike is detected, the event at  $T_n$  is dropped, the new event is scheduled at the appropriate time and the user is notified of the glitch. The glitch is not propagated because it usually signifies an error in the circuit design. Therefore, the simulation will continue as if an error did not occur and more meaningful information may be obtained about the correct operation of the circuit. This technique also reduces the amount of work the simulator is required to do since spikes represent activity in the circuit. Therefore, the overall CPU-time will be kept to a minimum by removing glitches from the simulation.

In SPLICE1.7, a fanout list (FOL) can only appear once on the time queue at any given time during the processing. This is a limitation for proper glitch handling, as will be seen in the pseudo-code description of glitch handling which follows. Two different problems are identified which are direct results

of the scheduling limitation.

#GLITCH HANDLER IN SPLICE1.7

PT = present time

$T_{next}$  = next time FOL is scheduled

$T_{last}$  = last time FOL was scheduled to be processed  
(or was actually processed)

if ( $T_{last} < PT$ ) { #node was processed in the past

store new\_state ;

schedule FOL at  $T_{next}$  ;

}

else if ( $T_{last} = PT$ ) { #node is scheduled now

if ( $T_{next} \geq T_{last}$ ) {

if (FOL processed) { # PROBLEM : glitch has been propagated

update new\_state ;

schedule FOL at  $T_{next}$  ;

}

else { #FOL has not been processed

#PROBLEM : cannot schedule FOL more than once

drop schedule at  $T_{last}$  ;

replace new\_state ;

schedule FOL at  $T_{next}$  ;

}

}

}

else if ( $T_{last} > PT$ ) { #node is scheduled in the future

if ( $T_{next} < T_{last}$ ) {

#reschedule time is earlier than originally scheduled time

report glitch ;

drop schedule at  $T_{last}$  ;

replace new\_state ;

schedule FOL at  $T_{next}$  ;

}

else if ( $T_{next} = T_{last}$ ) {

report glitch ;

replace new\_state ;

}

else if ( $T_{next} > T_{last}$ ) { #want to sched in future

report glitch ;

drop schedule at  $T_{last}$  ;

store new\_state ;

schedule FOL at  $T_{next}$  ;

}

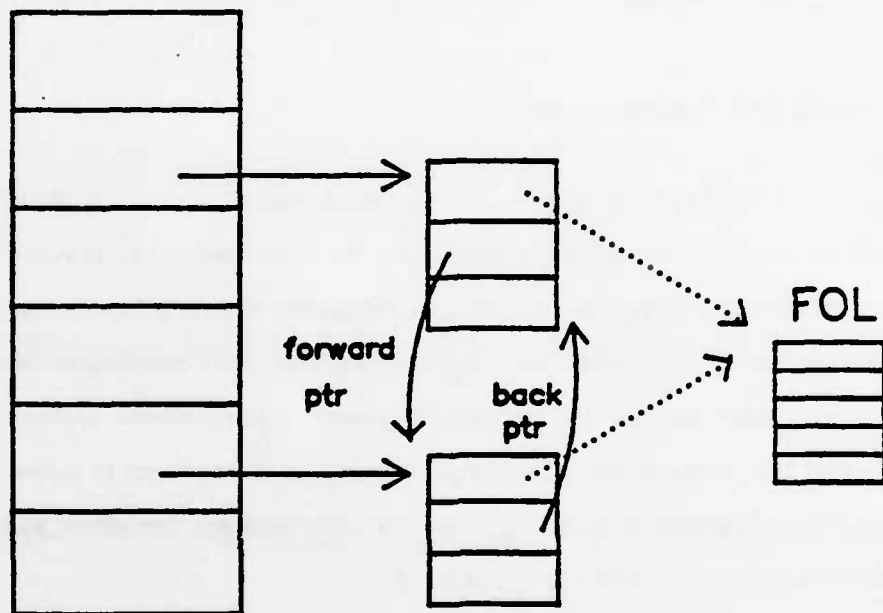
}

The problems identified above can be summarized as follows: depending on the order in which nodes are processed at a timepoint, the program may or may not propagate one particular type of glitch, which will be referred to as the Edge glitch or "E" glitch. Therefore, the output of the simulation depends on the order in which the circuit was specified by the user. The "E" glitch is always identified but its propagation is based on node processing order. One way to get around this problem is to use a two-pass approach by first performing a *leveling* operation [28] as a preprocessing step. This simply means that each node should be assigned a value based on its depth from the inputs. Then every node scheduled at a given timepoint should be processed in *ascending* order. This would incur some overhead but would produce the desired results, i.e., the same solution regardless of the order of the input description. At the present time, SPLICE1.7 will identify the glitch and may or may not propagate the glitch depending on the order the nodes are processed.

Another way to eliminate the problem is by modifying the scheduler data structure so that multiple schedules are allowed. Instead of scheduling FOLs, it would be better to schedule structures which point to the FOL. This structure would have to include other information such as the schedule time, and forward and backward pointers to the next and previous schedules of the same FOL in the time queue. This would allow easy access to all the schedules of a single FOL for adding and dropping subsequent events. This proposed data structure is shown in Fig. 3.8. One advantage of multiple scheduling is that the program can be modified to perform parallel fault simulation using this data structure.



time queue



**Fig. 3.8:** A proposed data structure to allow multiple FOL scheduling.

A simple circuit which is useful for debugging glitch handling code is the clock generator shown in Fig. 3.9. By adjusting  $t_r$  and  $t_f$  for each gate, all possible glitch conditions can be produced. For example, if  $t_r = t_f = 10\text{ns}$ , the "E" glitch can be generated.

### 3.5. Transfer Gate Modeling Issues

The incorporation of strengths into the state model does not in itself solve all the problems of MOS logic simulation. As described in the previous section, delay modeling is still difficult and the notion of strengths does not provide any leverage in solving the problem. Transfer gates complicate the situation even more because they introduce dynamic loading effects, bidirectional signal flow, node decay, and charge-sharing. In the sections to follow, these and other problems concerning the transfer gate are described and the solutions used in SPLICE1.7 are presented.

#### 3.5.1. Bidirectional Transfer Gates

In general, the transfer gate is a bidirectional element but it is usually found in a unidirectional application. That is, the designer intended signals to flow in one direction through the device. SPLICE1.7 provides unidirectional transfer gates (UTXG) for this purpose, as it simplifies the processing thereby reducing CPU-time.

On the other hand, there are occasions when transfer gates are used in bidirectional applications and therefore the logic simulator must be able to analyze them accurately. There have been a variety of modeling approaches for bidirectional transfer gates (BTXG), including the conventional approach of two unidirectional elements back-to-back as shown in Fig. 3.10. This

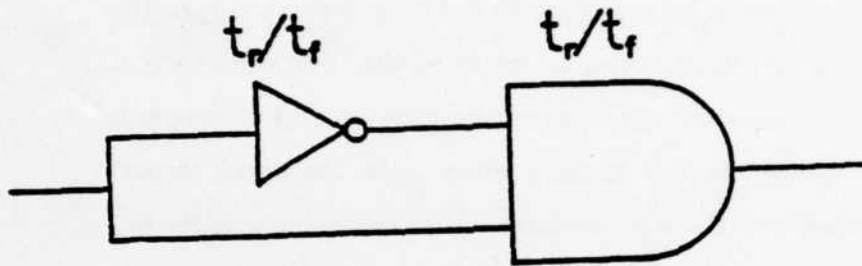


Fig. 3.9 : A simple circuit which can be used to generate the various glitch conditions by adjusting the values of  $t_r$  and  $t_f$ .

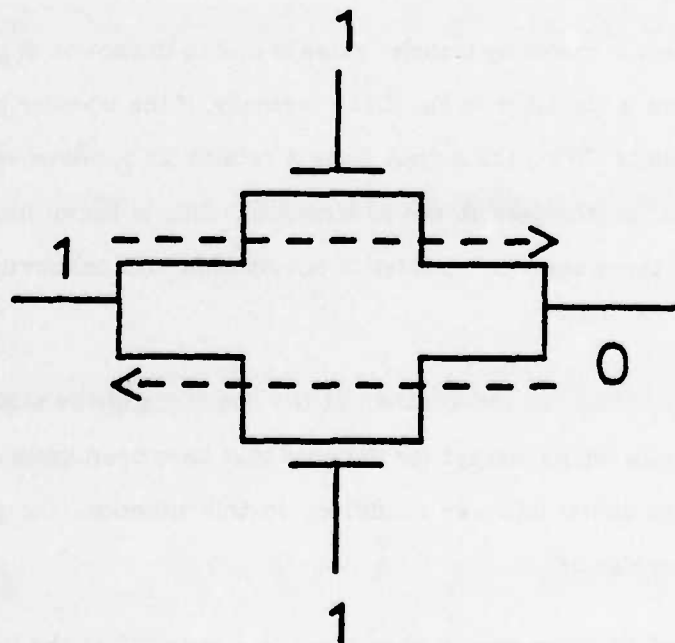


Fig. 3.10 : A bidirectional transfer gate model which employs back-to-back unidirectional transfer gates. The usual problem associated with this approach is that contentions may not be resolved properly.

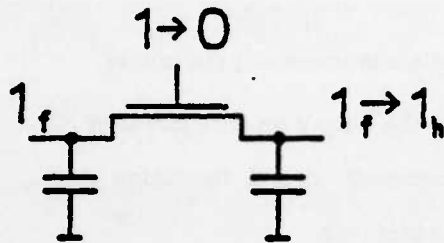
approach can lead to inconsistencies when different logic values are on opposite sides of the element, as is the case in Fig. 3.10. Each value can flow through the BTXG and reach the opposite side and these errors can percolate further through the circuit producing incorrect results. One simple way to process BTXG's in a consistent way is to introduce the concept of composite node relaxation (CNR). In this method, every node connected through transfer gates which are "ON" are considered to be the same node for processing purposes. All fanin lists for the composite node are combined into one list and a new state is determined based on the composite fanin list. Since all nodes connected by "ON" BTXG's are considered the same node, there is no delay between them.

### 3.5.2. Unknowns at Gate Inputs

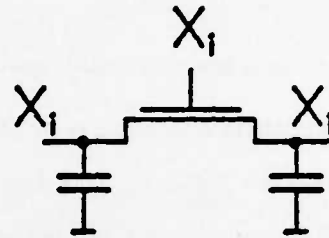
Another problem in modeling transfer gates is due to unknowns at gate inputs. The problem is identified in Fig. 3.11. Normally, if the transfer gate is "ON" and then shuts "OFF", the output Node A retains its previous value but is reduced in strength (goes to the  $H$  strength). This is shown in Fig. 3.11(a). There are three cases to consider in conjunction with unknowns at transfer gates.

**CASE 1 :** Fig. 3.11(b) indicates the situation at the beginning of the simulation. Virtually all gate inputs, except for the ones that have been initialized explicitly, are in the initial unknown condition. In this situation, the gate may or may not pass signals.

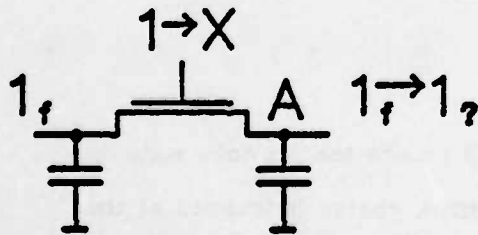
**CASE 2 :** The second situation occurs when there is a logic "1" at the input and it changes to a logic "X". In this case, the level at the output remains the same but the strength is not known. This is illustrated in Fig. 3.11(c).



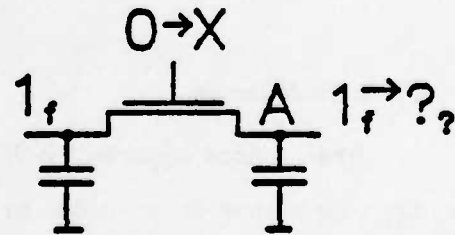
Normal operation with known  
states at gate input



Case 1: initial unknown produces  
unknown levels and strengths



Case 2:  $1 \rightarrow X$  produces an unknown  
strength at the output, although  
the level is known



Case 3:  $0 \rightarrow X$  produces an unknown  
level and strength

Fig. 3.11: Various cases when processing transfer gates with X at the gate input

**CASE 3 :** The third situation is the reverse of the second. Here, the input goes from logic "0" to logic "X". Both the value and strength may change. This is shown in Fig. 3.11(d).

There are a few alternative methods to handle unknowns at gate inputs.

- (1) a pessimistic approach is to generate  $X_f$  at the output so that it will be propagated further. This may produce incorrect circuit operation if CASE 2 is considered, but is the easiest to implement.
- (2) another approach is to have the notion of unknown strengths. Using this model, CASE 2 could be handled by setting the output node to its previous value with an unknown strength. This introduces some complications in the way the simulator processes nodes. Some bit pattern would have to be selected for unknown strengths. It is not clear how this special strength value would interact with other strengths.

Currently, method (1) is used in SPLICE1.7 and other methods are under investigation [29].

### 3.5.3. Node Decay

When a node acquires the  $H$  strength, it retains the previous state on the capacitance at the node. In physical terms, charge is trapped at the node but there are parasitic resistive paths from the node to ground or VDD. Therefore, the node will eventually lose its value and it will become unknown. This is referred to as *node decay*. The time constant for the decay is large but finite.

It is useful to include node decay as part of the simulation, especially for dynamic circuits. One way to do this is to detect the  $H$  strength at a node and schedule the node to decay after a specified amount of time by setting a

special flag at the node. If the node is not redriven before this time, the node is placed in the unknown state. If the node is redriven, the scheduled event would be dropped and processing would continue in the normal way. Unfortunately, the program would incur an excessive amount of scheduling and de-scheduling overhead, especially in the case of dynamic MOS circuits. Moreover, in SPLICE1.7, most of the scheduled nodes would be put into the pool (see Appendix II, part 5). The pool is an overflow area designed to store all schedules which are greater than 200 timepoints in the future. If node decay is processed as suggested above, the scheduled decay events would all be placed in the pool and eventually overflow the limit of the pool area. Clearly this is not a suitable approach.

An alternative approach, proposed by Boyle [30], is to simply store the decay time along with the node data structure and avoid scheduling altogether. Anytime the node is redriven, this value could be compared to the current time. If the current time is greater than the decay time, a warning message could be placed in a file, if the user has requested decay errors. Then processing would continue as if node decay had not occurred. It is not useful to simulate the circuit under decay conditions because it is usually a design error. Therefore, it is simply flagged as an error and then ignored for the remainder of the simulation.

### **3.6. Logic Simulation Implementation Details**

#### **3.6.1. General Program Flow**

The following is a high-level pseudo-code description of the general program flow during a logic analysis. Note the parallel between the ITA program



flow described in the previous chapter and the code below.

```

set all nodes to their initial values ;
schedule all FOL's at time 0 ; # FOL = fanout list
 $t_n = 0$  ;
while (  $t_n \leq TSTOP$  ) {
  for (each FOL in the queue at the current timepoint) {
    for (each element in the FOL) {
      for (each output node of an element) {
        process node ; #see next section for details
        schedule FOL if necessary ;
      }
    }
  }
  plot all active nodes ;
   $t_n \leftarrow t_{n+1}$  ;
}

```

### 3.6.2. Node Processing Details

```

# LOGIC NODE PROCESSING DETAILS
#
begin
  current_state  $\leftarrow$  (X,0) ;
  place node in CNL ; # CNL = composite node list
  for (each node in the CNL) {
    for (each element in the FIL) { # FIL = fanin list
      if (element = BTXG) & (gate = "ON") {
        place node in CNL ;
      }
      else {
        determine output_state (LS) of element ;
        intend_state  $\leftarrow$  output_state ;
      }
      if ( str(intend_state) > str(current_state) )
        current_state  $\leftarrow$  intend_state ;
      else if ( str(intend_state) = str(current_state) )
        if ( lev(intend_state)  $\neq$  lev(current_state) )
          lev(current_state)  $\leftarrow$  X ;
    }
  }
  new_state  $\leftarrow$  current_state
  if (new_state  $\neq$  old_state) {
    for (each node in CNL) {
      calculate delay(old_state,new_state) ;
      call GLITCH HANDLER to schedule FOL ; # FOL = fanout list
    }
  }
end

```

### 3.7. Switch-level Simulation

The definition of UTXG and BTXG elements (given in 3.5.1) allows switch-level simulation [9,10] to be performed using SPLICE1.7. Loads can be modeled using a UTXG with a "weak" output strength. Drivers can be modeled using a UTXG with a "forcing" output strength. Either a BTXG or a UTXG can be used for pass transistors depending on the application. Other floating transistors must be BTXG's. If "ton" and "toff" are specified as 1 unit of time, then a unit-delay switch-level simulation will be performed by SPLICE1.7. For obvious reasons, delays at the switch-level cannot be modeled in the same way as it is currently done in SPLICE1.7 for standard Boolean gates. Two approaches have been proposed to introduce detailed timing information at the switch-level using a resistive simulation model [31,32]. These methods are under investigation at the present time.

## CHAPTER 4

### 4. Examples and Results

In this chapter, a number of simulation results and program performance statistics of SPLICE1.7 are presented. Five aspects of the program are examined in the sections to follow. These are :

- the program performance statistics such as processing speed for the electrical and logic analyses, typical storage requirements per element, iteration counts, etc.
- the identification of bottlenecks using profilers
- the factors which affect the run-times such as **mindvsch**, **sor**, **mrt** and floating capacitors
- SPLICE1/SPICE2 comparisons for execution speed, memory requirements and simulation accuracy
- mixed switch, logic and electrical-level simulations

The simulations were carried out on the following circuits:

- (1) **Digital Filter Circuit** : This circuit was obtained from [1]. It is the control logic for a digital filter circuit. There are 705 MOS transistors and 393 nodes in the circuit. The simulation period is  $4\mu\text{s}$ .
- (2) **Counter-Decoder-Encoder Circuit** : This circuit is a combination of a 4-bit counter driving a 4:16 decoder and a 16:4 encoder. It is referred to as the CDE circuit in the rest of the chapter. The switching characteristics were based on the specifications provided in a TTL Handbook [33]. The circuit has 1,326 MOS transistors and 553 nodes and is the largest

circuit simulated so far. The simulation period is also  $4\mu s$  [34].

- (3) **NMOS Operational Amplifier** : This circuit was obtained from [35]. It was designed as part of a phase-locked loop circuit. This circuit is used to illustrate the capability of ITA when simulating analog circuits.
- (4) **Boot-strapped Inverter Circuit** : This circuit was described earlier in Chap. 2. It is illustrated in Fig. 2.4(b). The circuit is used to examine the effects of a floating capacitor element in an ITA simulation.
- (5) **Industrial Microprocessor Control Circuit** : This is the critical path through the control circuitry of a  $\mu P$  designed using NMOS technology.
- (6) **Industrial 64K Ram Circuit** : This is a portion of a high-speed 64K static RAM circuit designed using CMOS technology.
- (7) **4 x 5 Multiplier Circuit** : This circuit uses standard multiplier structure. It features novel exclusive-OR and ADDER functions designed by Kuninobo [38]. This example is used to illustrate mixed-mode simulation and to compare the run-times associated with transistor-based simulation at the electrical and switch levels.

#### **4.1. Program Performance Statistics**

In order to predict the run-times and memory requirements of the program SPLICE1.7, the program execution speed and memory usage statistics are required. These statistics have been tabulated below for both the electrical and logic simulators.

### Electrical Simulation Statistics

Node Evaluations	400 nodes/sec.
SOR-Newton Iterations (no floating caps)	3-5 iterations/node
SOR-Newton Iterations (with floating caps)	6-20 iterations/node

### Electrical Element Storage Requirements

Elements	Type	Words Required
Transistors	Load	3 x no. of loads
	Driver	4 x no. of drivers
	Transistor	5 x no. of transistors
Capacitors	Grounded	0
	Floating	3 x no. of capacitors
Resistors		3 x no. of resistors
Element Model	Type	Words Required
Transistors	Load	13 x no. of loads
	Driver	13 x no. of drivers
	Transistor	14 x no. of transistors
Capacitors	Grounded	1 x no. of grounded capacitors
	Floating	3 x no. of floating capacitors
Resistors		3 x no. of resistors

### Logic Simulation Statistics

Node Evaluations 650 nodes/sec.

### Logic Element Storage Requirements

Elements	Words Required
inverter	3 x no. of inverters
buffer	3 x no. of buffers
AND	~ 5 x no. of ANDs
OR	~ 5 x no. of ORs
NAND	~ 5 x no. of NANDs
NOR	~ 5 x no. of NORs
XOR	~ 5 x no. of XORs
XNOR	~ 5 x no. of XNORs
transfer gates	~ 4 x no. of devices
Model Type	Words Required

inverter	11 x no. of different inverter models
buffer	11 x no. of different inverter models
AND	~ 11 x no. of different AND models
OR	~ 11 x no. of different OR models
NAND	~ 11 x no. of different NAND models
NOR	~ 11 x no. of different NOR models
XOR	~ 11 x no. of different XOR models
XNOR	~ 11 x no. of different XNOR models
transfer gates	~ 8 x no. of device models

### Node Storage Requirements

$N$  = number of circuit nodes

Data	Logic Node	Electrical Node
Node list	1	1
Node pointers	$N$	$N$
Node data	$8N$	$9N$
Node FIL	$N$	$3N$
Node FOL	$3N$	$3N$
Capacitor	$N$	$N$

### 4.2. Profile Statistics

The SPLICE1.7 program execution times can be reduced somewhat by applying the techniques suggested in Chap. 2. These were based on intuitive arguments. It is important to identify bottlenecks in the program and identify where it is spending most of its time in a quantitative way. A profiler is a modern programming tool which is very useful for this task. It monitors the program during execution and provides information relating to the percentage of time spent in each subroutine. Using this information, the program can be modified in sections where it will provide the most benefit.

The following profile statistics were obtained from a simulation of the digital filter circuit using electrical analysis.

total time: 3196 seconds

time(%)	time(sec.)	no. of calls	name	subroutine task
28.7	916.33	1222883	prtim	processing a timing node
17.3	554.47	2449814	tntxg	evaluate transistor model
14.2	455.32	5799795	getxv	get a value from another node
8.2	231.63	951895	sqr	perform a square root operation
6.3	203.63	900167	tndri	evaluate driver model
5.1	164.07	658140	tnloa	evaluate load model
4.7	152.62	792936	prelm	process an element in the FOL
1.8	51.92	4001	prfot	process a FOL in the time queue
1.3	40.23	276985	adsfo	add a FOL to the time queue
1.2	37.53	24046	dropf	drop a FOL from the time queue
0.3	9.70	20547	prout	print out a node

It is clear that most of the time is spent processing nodes and evaluating transistor models for the SOR-Newton iteration. Therefore, any speed-up techniques should be applied to these areas of the program. It is expected that, as the program is developed further, information provided by the profiler can be used to significantly reduce execution time, particularly for electrical simulation. Other methods, currently available to the user to reduce the total run-time, are described in the next section.

#### 4.3. Factors Affecting Execution Time in Electrical Simulation

##### 4.3.1. CPU-time vs. MRT

SPLICE1.7 has a user-specified fixed minimum timestep called the **mrt** (Minimum Resolvable Time). The symbol  $h$  will be used to refer to the timestep associated with a particular node. In SPLICE1,  $h$  is some integer multiple of **mrt**. Although the minimum timestep is fixed, the value of  $h$  for a specific node is dependent on the activity at that node. For example, when the node is active,  $h$  is equal to **mrt**. Otherwise,  $h$  is defined by the time difference between two events at the node. In a sense, the timestep at a node is determined implicitly by the activity in the circuit.



Since there is no explicit timestep control mechanism in SPLICE1, the CPU-time required to perform electrical simulation is a strong function of *mrt*. Fig. 4.1 illustrates this relationship for the CDE circuit. There is an optimum value of *mrt* for this particular circuit at 1ns. At values of *mrt* greater than the optimum, the program iterates longer to produce solution at a particular timepoint. This is due to the fact that a linear prediction generates a poor guess as the timestep is increased and the diagonal terms of the conductance matrix,  $\frac{C}{h}$ , are reduced thereby weakening the diagonal dominance property of the matrix. In fact, at a very large value of *mrt* the program may not converge at all.

At *mrt* values less than the optimum, the program is forced to do more work during the active periods than is really necessary, based on the time constants in the circuit. Therefore, there is a rapid rise in the curve in Fig. 4.1 below the optimum. It has been observed that at very small timesteps, the curve begins to level off. This is probably due to the fact that the prediction step is very accurate and only one or two iterations are required for convergence.

The relation between CPU-time and *mrt* suggests that, for a given technology, the optimum value should be obtained through experiment and used in all further simulations. Of course, if an explicit dynamic timestep control mechanism is implemented, this would not be necessary.

#### 4.3.2. CPU-time vs. MINDVSCH

In SPLICE1, events at a node are propagated to its fanouts if the change in the node voltage is considered to be significant. If the change is not significant, the fanouts are not scheduled and the node is returned to its

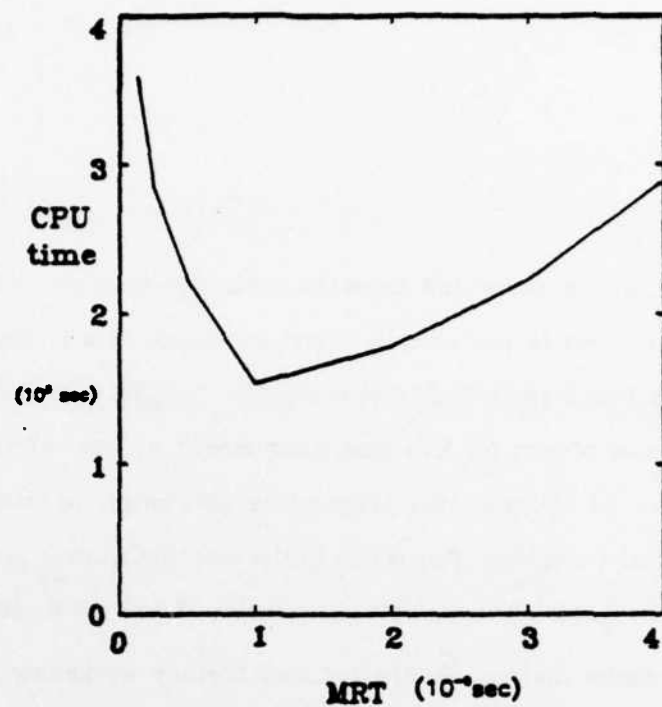


Fig. 4.1 : CPU-time vs. MRT

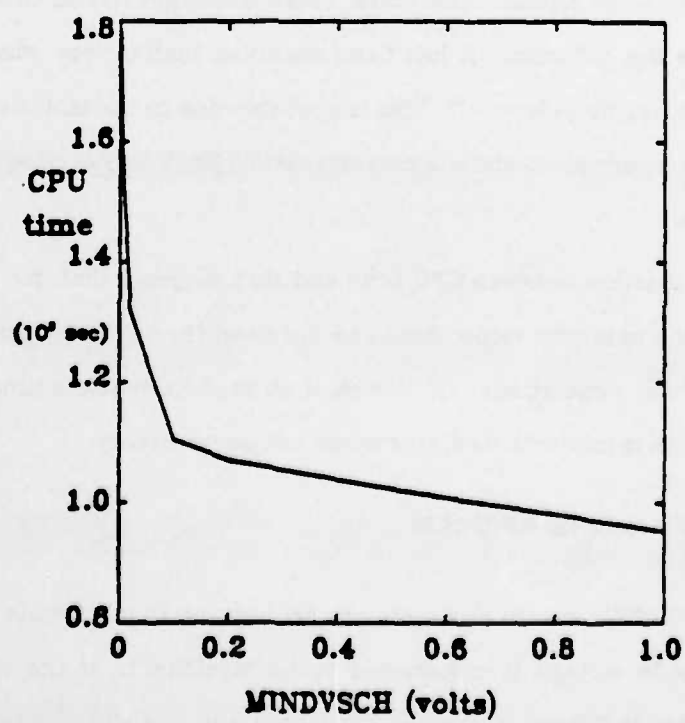


Fig. 4.2 : CPU-time vs. MINDVSCH

original value. This constitutes the event-driven selective-trace feature in SPLICE1. The scheduling threshold parameter, used to determine whether or not the change is significant, is called `mindvsch`. It is specified in units of volts, by the user, for an entire simulation and is the same for every node in the circuit.

The value chosen for `mindvsch` has a profound effect of the simulation results. Careful consideration must be given to select an appropriate value for this parameter. It can be thought of as the minimum voltage change which can affect the fanout elements of a node. Based on experience with the program, an appropriate value for most digital circuits is  $1mV$ . It is much smaller for analog circuits, particularly if there are high-gain stages.

The effect of `mindvsch` on CPU-time is quite dramatic as shown in Fig. 4.2. As `mindvsch` is increased, the CPU-time goes down. This suggests that the value should be made as large as possible. Unfortunately, some significant events may be accidentally dropped if the `mindvsch` is too large resulting in a loss of accuracy. Also, node voltages can only reach a value which is within `mindvsch` of their final value because the remaining voltage change is not considered significant. Hence, if `mindvsch` is too large, there will be errors at the end of each transition.

#### 4.3.3. Effect of Floating Capacitors

As indicated in Chap. 2, floating capacitors no longer pose a problem to the electrical analysis in terms of accuracy but tend to degrade the simulator performance. The factor which determines the amount of degradation is the ratio of the floating capacitor to the grounded capacitor. In order to illustrate the relationship between CPU-time and capacitance ratio, the boot-strapped inverter of figure 2.4(b) was simulated with different values of

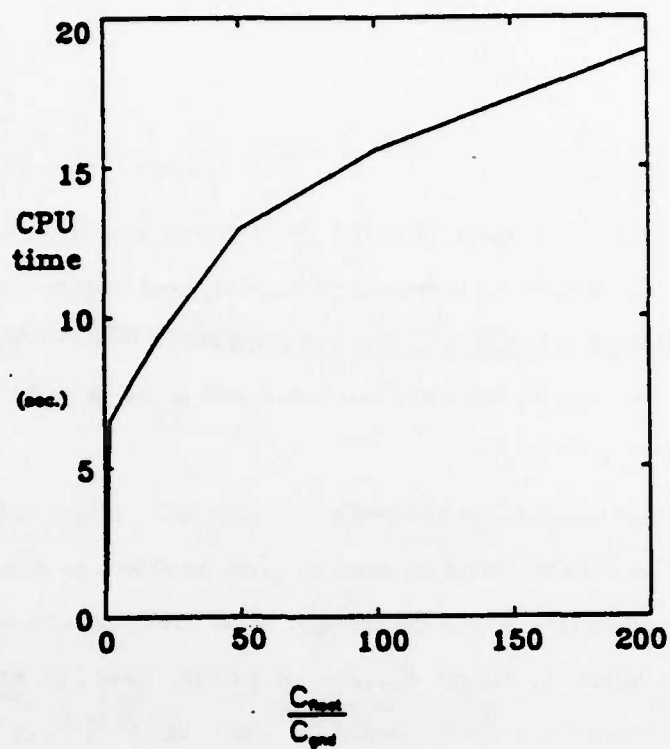


Fig. 4.3 : CPU-time vs. Capacitance Ratio

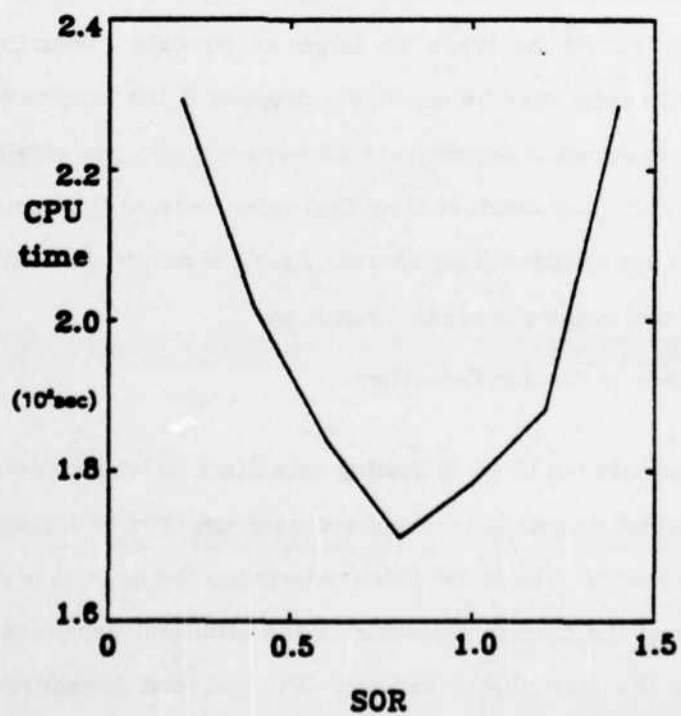


Fig. 4.4 : CPU-time vs. SOR

$\frac{C_{float}}{C_{gnd}}$ . The results have been plotted in Fig. 4.3. It is clear from this graph that the relationship obeys a square-root law.

Although the graph indicates that solutions may be obtained regardless of the ratio, this is not true in general. Therefore, if the ratio is too large, the iteration may not converge. Special techniques must be used to reduce the number of iterations required to solve nodes with floating capacitors since this is usually the case. Research is underway to find ways to accomplish this.

#### 4.3.4. CPU-time vs. SOR

The `sor` parameter was introduced in Chap.2 as an acceleration parameter for the nonlinear Gauss-Seidel iteration. This parameter has a significant effect on the CPU-time but does not affect simulation accuracy. Fig. 4.4 shows the relationship between CPU-time and `sor` obtained from simulations performed on the digital filter circuit. There is an optimum value of `sor` which minimizes the run-time of the simulation. In this case, the optimum value is 0.8.

Although the optimum value changes from technology to technology, it is worthwhile to obtain the value experimentally as it may provide a substantial improvement over the standard Gauss-Seidel iteration.

#### 4.4. SPICE2 vs. SPLICE1.7

Five circuits were simulated at the electrical level using SPICE2 and SPLICE1.7 to compare run-times, memory requirements and accuracy. These simulations were performed on a VAX-11/780 under the UNIX operating system. In both simulators, default parameter values were used for the

convergence criteria, integration method, and accuracy tolerance. The values are available in the user guide for each program.

#### 4.4.1. CDE Circuit

A block diagram of the CDE circuit is shown in Fig. 4.5. The details of the blocks are given in [33]. This circuit is large and highly unidirectional in nature. A 4-bit counter provides a sequence of inputs to the 4:16 decoder, the output of which is encoded to 4-bits. The simulation period was  $4\mu\text{s}$  with an *mrt* of 1ns. As shown in Fig. 4.6, the output waveforms have long latent periods. For this circuit, SPLICE1.7 was 86 times faster than SPICE2 and its memory usage was 35 times smaller, for comparable accuracy. More importantly, the SPICE2 simulation required 32 hours whereas the SPLICE1 simulation required only 40 minutes! This represents a *substantial* improvement in speed and allows a simulation of this magnitude to be quite feasible. A small fraction of this speed advantage can be attributed to the fact that SPLICE1 has been tailored for transient analysis, but the key reason for the improvement is the efficient exploitation of latency.

The simulation results are summarized in Table 4.1. Also included in this table are the simulation results obtained using SPLICE1.3, an earlier version of SPLICE1 which used NTA. It is interesting to note that SPLICE1.7 required only twice as much time as SPLICE1.3 to produce a solution even though it iterates to convergence. SPLICE1.3 uses only one SOR-Newton iteration at each timepoint, but it can reject the new solution if the change in voltage over an *mrt* is considered to be too large, as determined by a parameter called "maxdvstep". This is done to maintain simulation accuracy. For example, if the voltage change between times  $t_{n-1}$  and  $t_n$  is con-

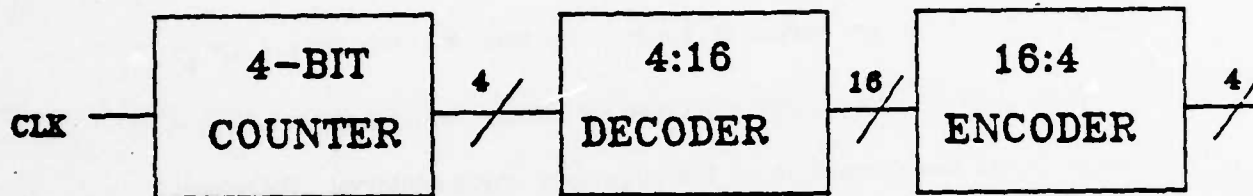


Fig. 4.5 : Block Diagram of the CDE circuit

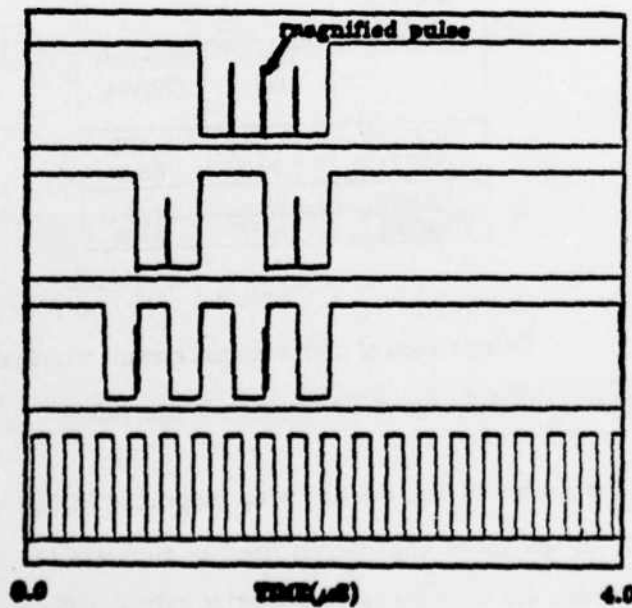


Fig. 4.8 : Output waveforms of CDE circuits. The waveforms feature long latent periods which is an advantage for ITA.



sidered to be too large, the program would cut the timestep by a factor of 4 and perform another series of single iterations at timepoints  $t_{n-1} + \frac{h}{4}$ ,  $t_{n-1} + \frac{h}{2}$ ,  $t_{n-1} + \frac{3h}{4}$  and  $t_n$ . Further timestep cutting would be done if the voltage change was deemed to be too large over any subinterval. Therefore, many evaluations may be performed to produce the solution at a timepoint, although a single iteration is always used at any given point in time. The attempt to maintain accuracy in this manner increases the overall simulation time. Furthermore, on the first iteration at a timepoint, SPLICE1.7 uses previous history to predict a new voltage at a node, thereby reducing the total number of iterations required to converge to a solution.

Circuit Mosfets Nodes	CDE 1,328 553	
	Time (s)	Memory (Kbyte)
SPICE2G	115,840	2,420
SPLICE1.7	1.740	68.9
Ratios	66	35
SPLICE1.3	843	68.9

Table 4.1

Comparison of conventional circuit simulation  
and ITA for the CDE circuit.

One of the pulses in Fig. 4.6 has been magnified in Fig. 4.7 to compare the accuracy of the three approaches used to simulate the CDE circuit, as indicated in Table 4.1. The pulses from SPLICE1.7 and SPICE2 are centered at 1627ns and 1629ns respectively. This difference would be indistinguishable at the level shown in Fig. 4.6 and it is not clear which is the more accurate solution. In this case, the true solution lies between the SPLICE1.7 and

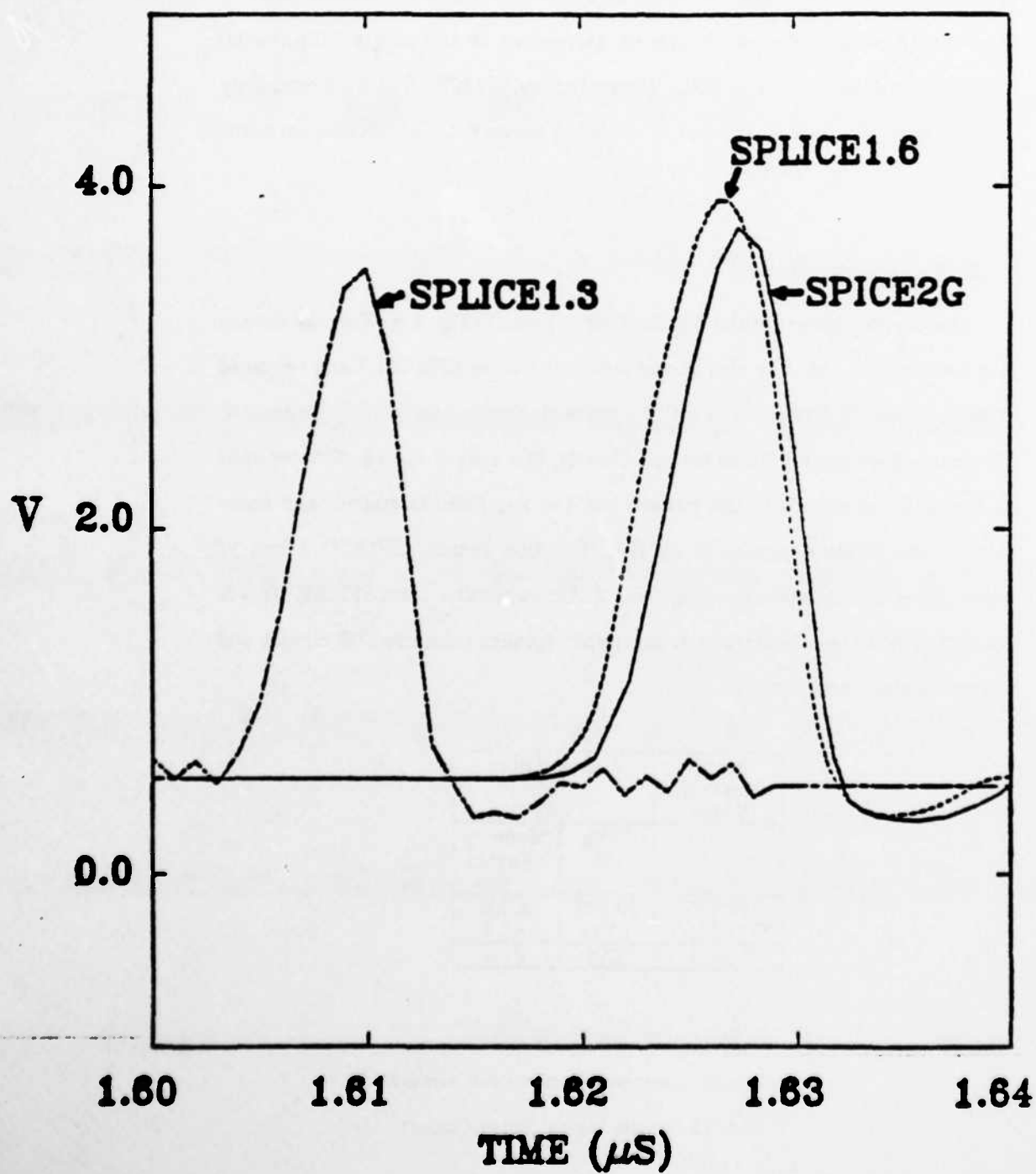


Fig. 4.7: The pulse shown in 4.6 has been magnified here to compare ITA with NTA and SPICE2.

SPICE2 results. The output of SPLICE1.3 features some numerical noise in the vicinity of 0.5 volts which can be attributed to the single SOR-Newton iteration used in NTA. The pulse generated by SPLICE1.3 is approximately correct in its size and shape but is centered incorrectly at 1608ns, an error of 20ns.

#### 4.4.2. Digital Filter Circuit

The block diagram of the digital filter is given in Fig. 4.8. Further details may be found in [1]. The circuit was simulated using SPLICE1.7 and required 1783 seconds. The original SPLICE1 program, as described in [1], required 453 seconds which is 4 times faster. Clearly, the cost of ITA vs. NTA depends on the size and nature of the circuit, but the key point is guaranteed accuracy in the solution produced by ITA. For this circuit, SPLICE1.7 was 17 times faster and its memory usage was 21 times smaller than SPICE2. This is due to the fact that this circuit is somewhat smaller than the CDE circuit and has much more activity.

Circuit Mosfets Nodes	Digital Filter	
	705 393	
	Time (s)	Memory (Kbyte)
SPICE2G	30,582	1,038
SPLICE1.7	1,783	48.4
Ratios	17.1	21.4

Table 4.2  
Comparison of conventional circuit simulation,  
and ITA for the Digital Filter Circuit

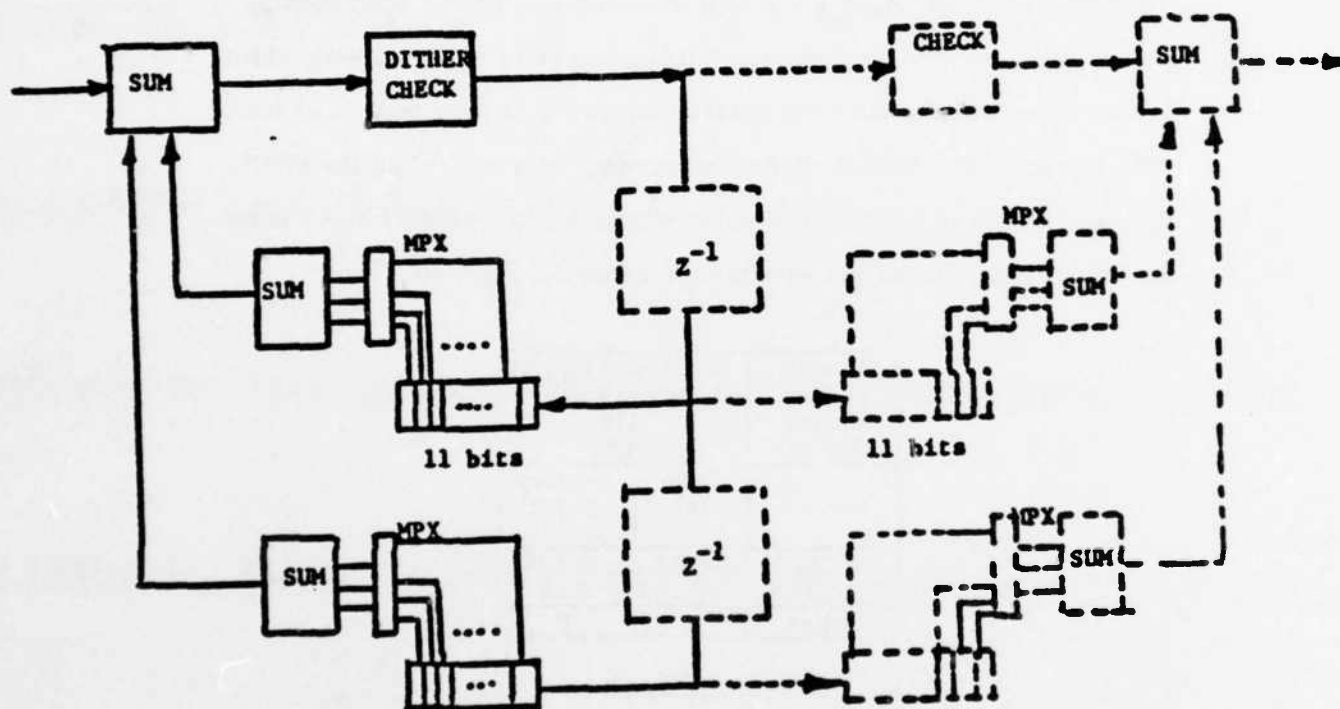


Fig. 4.8 : Block Diagram of Digital Filter Circuit

#### 4.4.3. Industrial $\mu$ P Control Circuit

This schematic for this circuit is shown in Fig. 4.9. It contained over 100 transistors and 100 diodes and is representative of a typical simulation performed using the SPICE2 program. Although no extra elements were added to this circuit, there was a capacitance to ground at each node of at least 10FF in value. The SPLICE1.7 simulation required 4 min. while the SPICE2 simulation required 24 min. The memory requirements were 8 times less for the SPLICE1 job. The output waveforms are shown in Fig. 4.10.

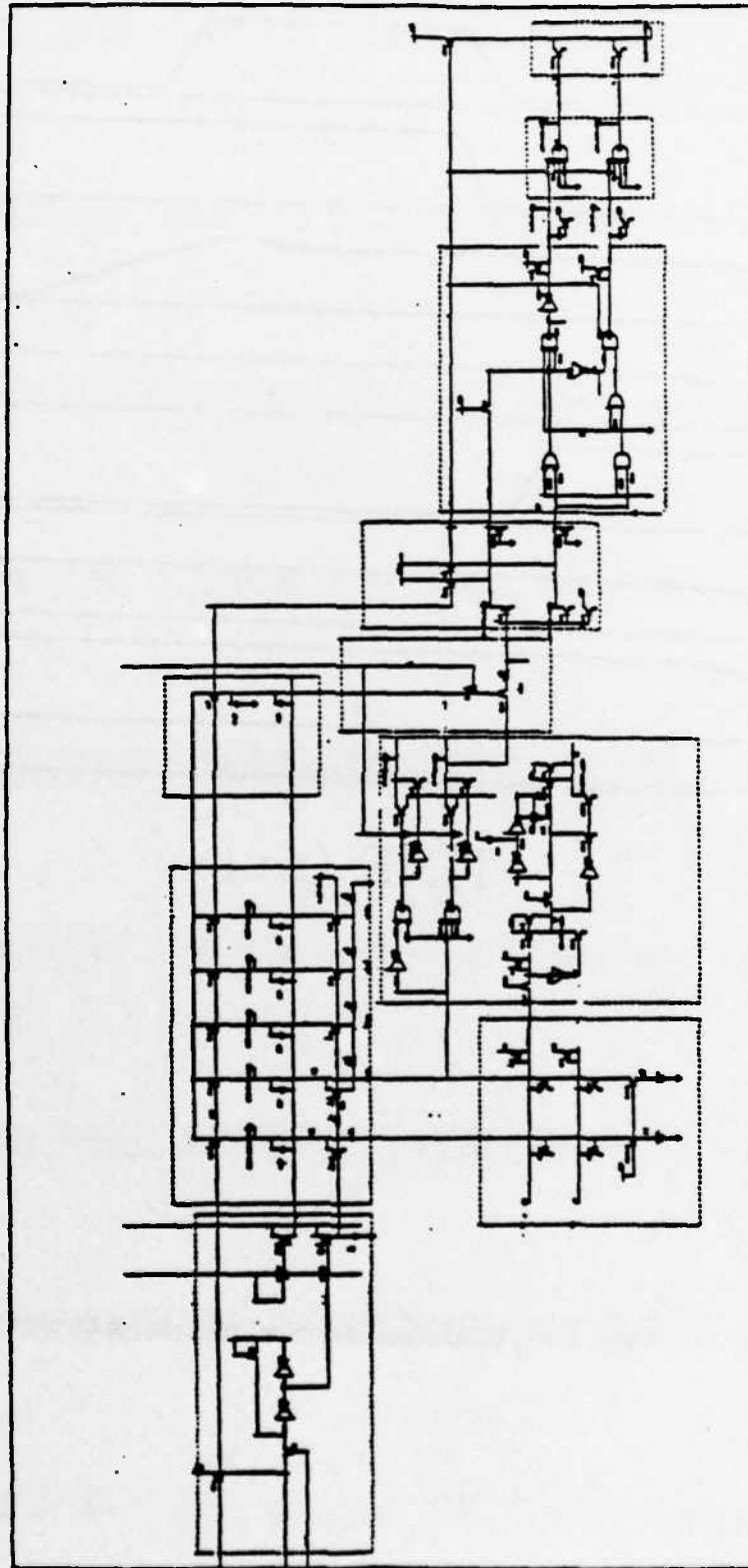
Circuit	uP Control Circuit	
Mosfets	116	
Diodes	116	
Nodes	86	
	Time (s)	Memory (Kbyte)
SPICE2G	1426.8	205.9
SPLICE1.7	177.2	26.2
Ratios	8	8

Table 4.3

Comparison of conventional circuit simulation,  
and ITA for an Industrial  $\mu$ P Control Circuit

#### 4.4.4. Industrial 64K CMOS Static RAM Circuit

The block diagram for this example is given in Fig. 4.11 and each subcircuit schematic is shown in Fig 4.12. This circuit contained over 300 transistors and is an example of a industrial circuit which would be very expensive to simulate using SPICE2. The circuit contained only 38 explicit grounded capacitors out of 151 nodes. Diodes were used on the remainder of the nodes to model the parasitic junction capacitance effects. This was a sufficient condition to obtain convergence at every timepoint.



**Fig. 4.9 : Worst-case path through an Industrial Microprocessor Control Circuit**

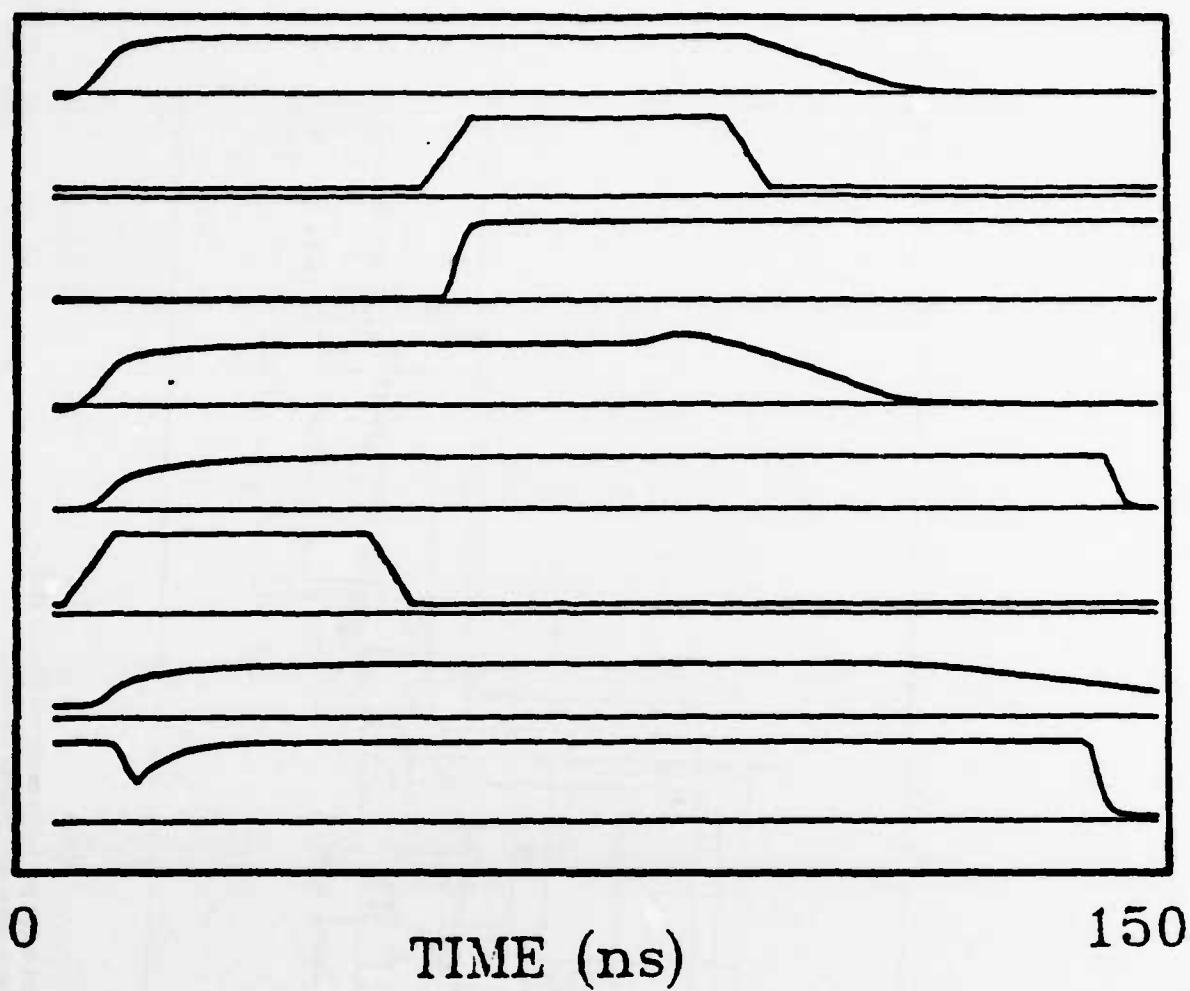


Fig. 4.10 : Output waveforms of uP Control Circuit



In this case, SPICE2 required approximately 3 hours to produce a solution whereas SPLICE1.7 required only 10 minutes. A comparison of the output waveforms is given in Fig. 4.13. The results are very close in all cases except in a few instances where SPICE2 exhibits point-to-point ringing. This is a product of the trapezoidal integration method used by default in SPICE2, which allows it to take larger timesteps but may cause ringing if the timestep is too large. SPLICE1.7 uses a Backward-Euler integration scheme, and for this method no numerical ringing is present in the waveforms.

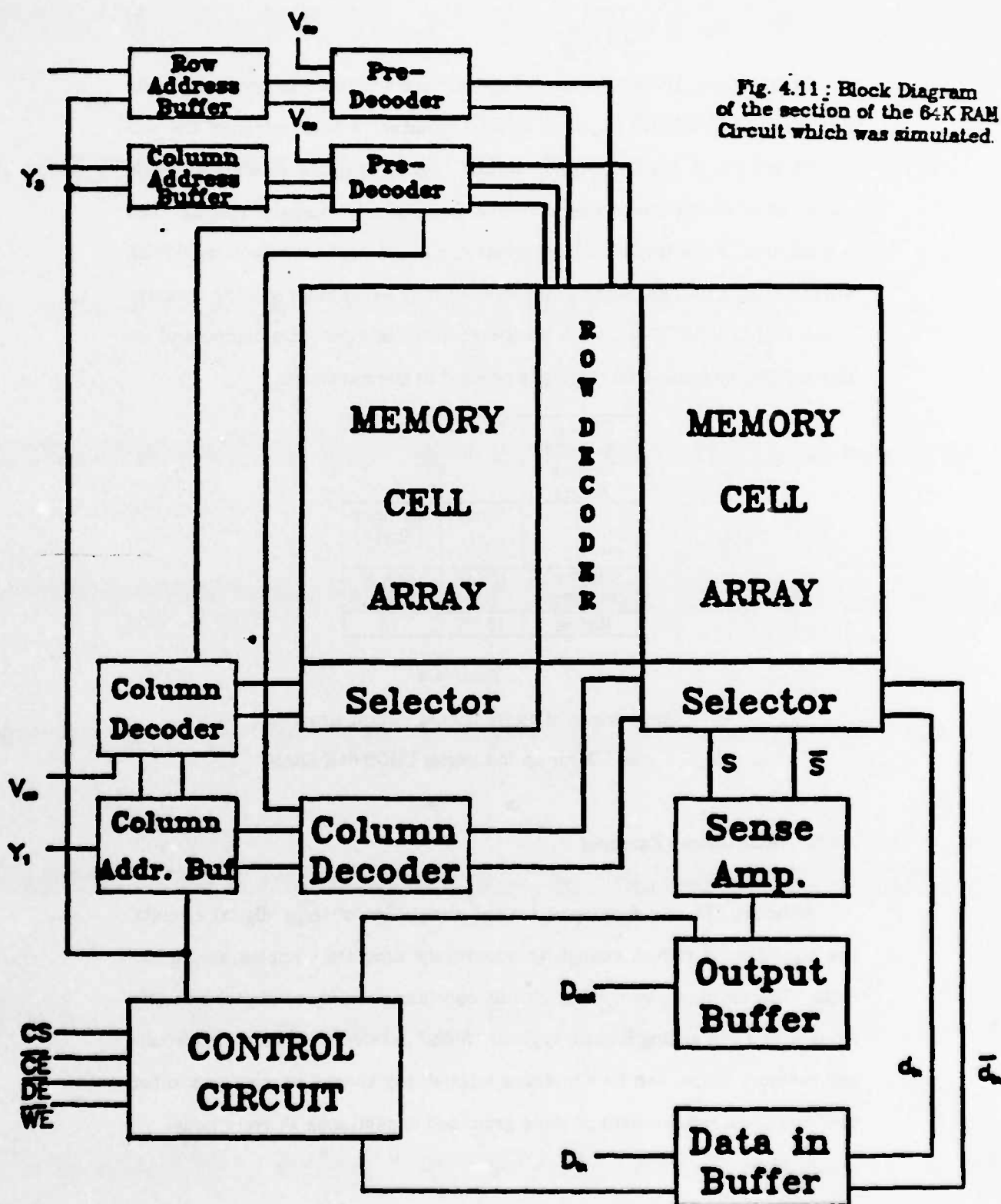
Circuit	64K CMOS SRAM	
Mosfets	344	
Diodes	277	
Nodes	151	
	Time (s)	Memory (Kbyte)
SPICE2G	10448	506.3
SPLICE1.7	623	49.9
Ratios	16.75	10

Table 4.4

Comparison of conventional circuit simulation,  
and ITA for an Industrial CMOS 64K SRAM

#### 4.4.5. NMOS OpAmp Example

Although ITA was developed for the simulation of large digital circuits, the algorithm is robust enough to accurately simulate complex analog circuits. Therefore, an integrated circuit consisting mainly of digital circuitry along with a few analog blocks, typically found in telecommunication circuits and memory chips, can be simulated without any special precautions, other than the usual requirement of some grounded capacitance at every node.



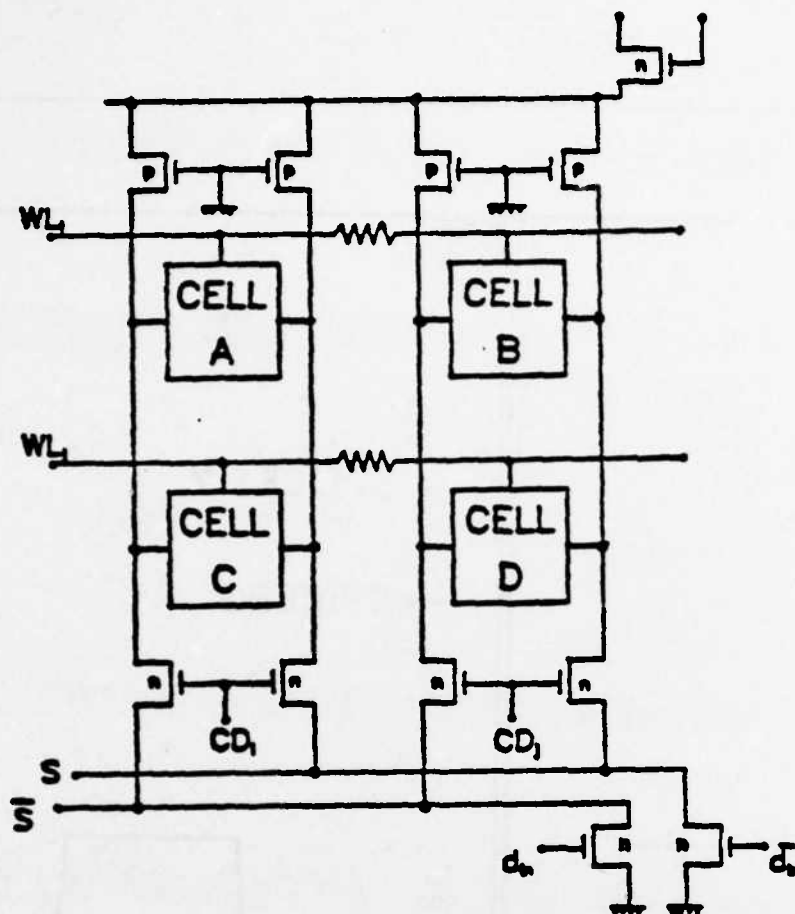


Fig. 4.12(a) : Memory Cell Array Details

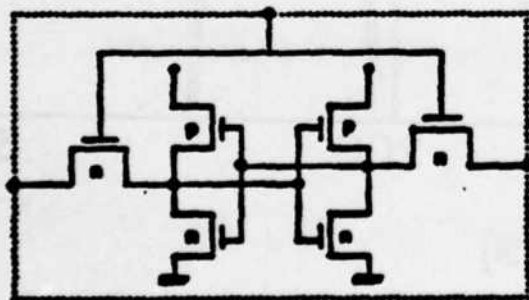


Fig. 4.12(b) : Memory Cell Box

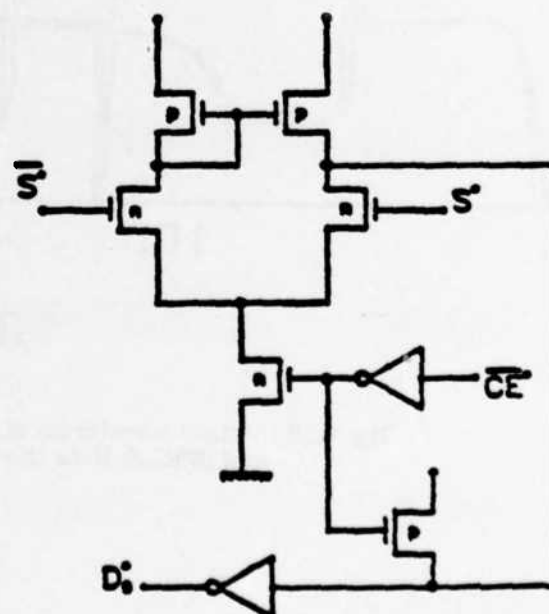


Fig. 4.12(c) : Sense Amplifier Circuit

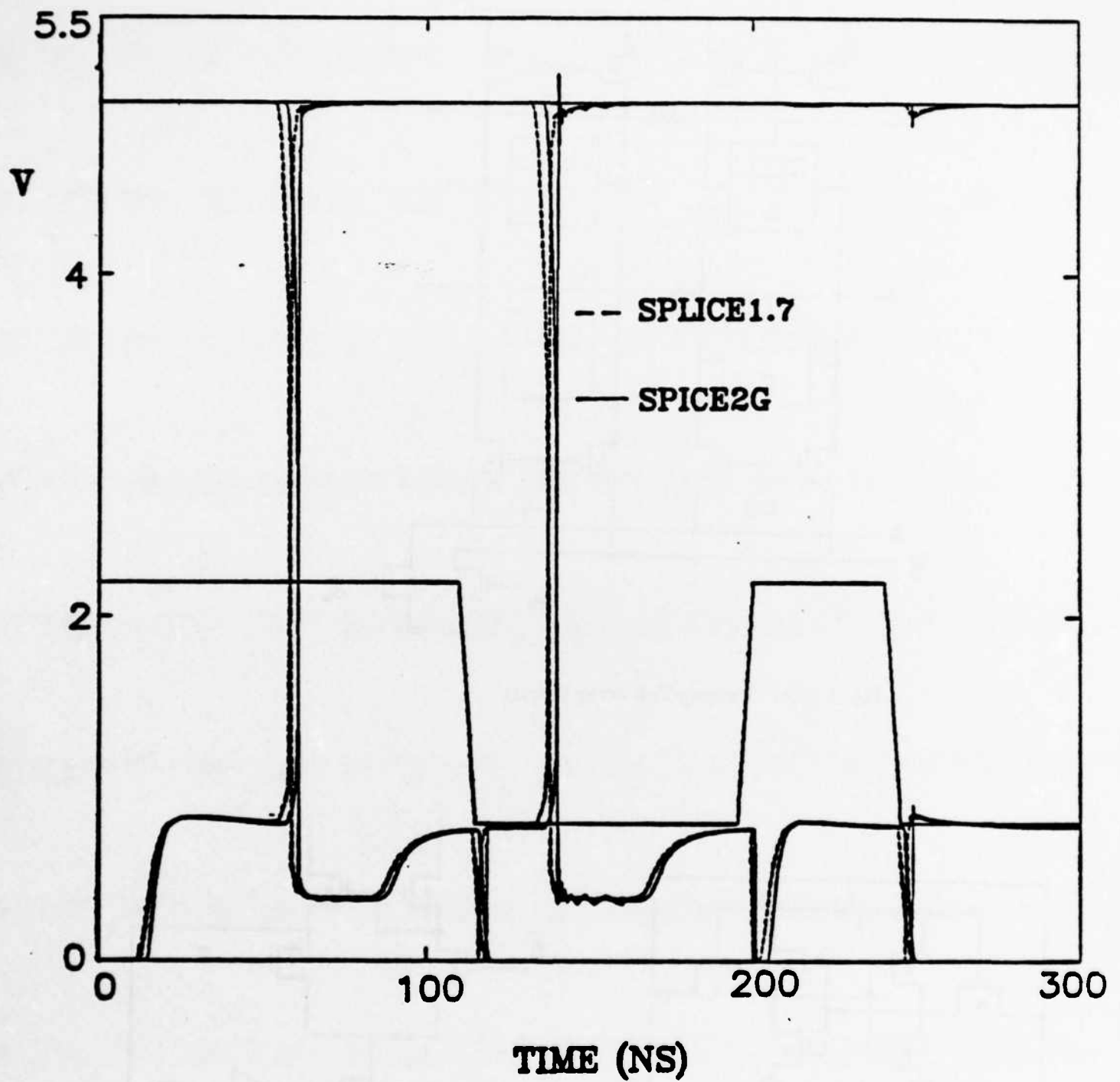


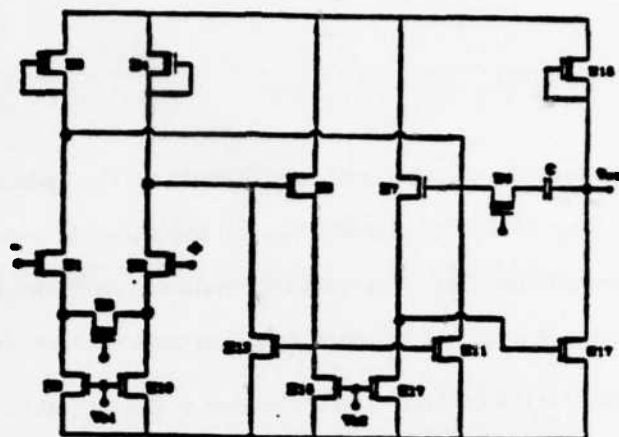
Fig. 4.13 : Output waveforms of 64K RAM circuit from SPLICE1 and SPICE2. Note the ringing produced in the SPICE2 output.

To illustrate the capability of the ITA method, the OpAmp in Fig. 4.14 was simulated using SPLICE1.7 and SPICE2. All the parasitic capacitances associated with each transistor were fully represented. As shown in the schematic diagram, there is a large 10pF compensation capacitor providing a capacitive feedback path in the circuit. The transistor at the output is  $\frac{360\mu}{8\mu}$  to provide high gain at the output node. The circuit was connected in a unity-gain configuration and a step voltage was applied at the input. Fig. 4.15 is a comparison of the outputs of SPLICE1.7 and SPICE2. The results are identical except in the neighborhood of time  $t=0$  due to slightly different initial conditions assumed by each program. However, the execution time of SPICE2 was two times faster than SPLICE1.7 because of the size and nature of the circuit. It is expected that this difference will be reduced as the program is developed further.

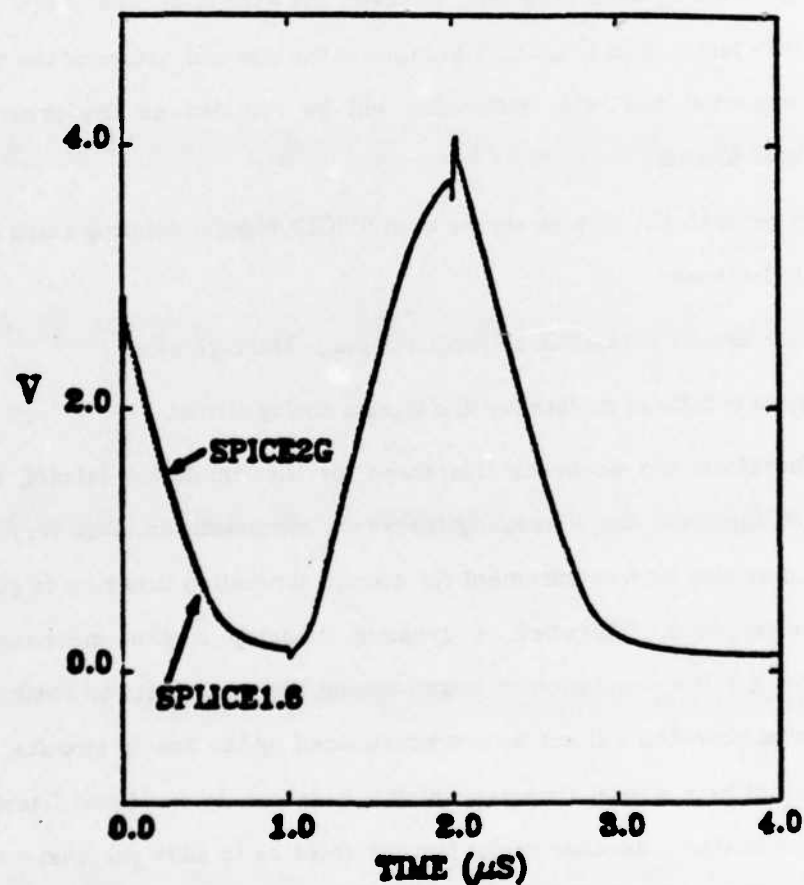
In general, ITA may be slower than SPICE2 when simulating small analog circuits because :

- they usually contain large feedback paths and high gain
- there is little or no latency in a typical analog circuit.

Therefore, the accuracy tolerances for the simulation (*abstol*, *reltol*) must be tight and the scheduling threshold, *mindvsch*, must be very small. There may also be a requirement for a small simulation timestep to guarantee convergence. Therefore, a dynamic timestep control mechanism is essential for the simulation of mixed analog/digital circuits to ensure that the global timestep will not be overconstrained by the analog circuits. Each node could have a local timestep which is based on its own Local Truncation Error estimation. Another useful feature would be to allow parameters such as *abstol*, *reltol* and *mindvsch* to be specified on a per-node basis. In this



**Fig. 4.14 : NMOS Operational Amplifier Circuit.** This circuit features tight coupling between nodes and high forward gain (due to large output devices) and large capacitive feedback (due to the 10pF compensation capacitor).



**Fig. 4.15 : Output Waveforms of OpAmp circuit from SPICE1 and SPICE2.** The results are indistinguishable illustrating the accuracy of the ITA method.

way, certain nodes would be forced to iterate longer than others to ensure accuracy at these nodes. These and other techniques may be used to improve the performance of the simulator for handling analog circuits, although ITA is not ideally suited to the task.

#### 4.4.6. CPU-time vs. Circuit Size

The run-times for the circuits described in this section are plotted against the circuit size in Fig. 4.16. It is clear from this plot that SPLICE1.7 is much faster than the SPICE2 program for large circuits. In fact, as the circuit size increases, the improvement factor increases. This is due to the fact that the linear equation solution time in SPICE2 increases rapidly with the circuit size, as described in Chap. 2. The run-time in SPLICE1.7 is proportional in the activity in the circuit and the ~~mrt~~ rather than circuit size. If the circuit is small, the standard approach is usually more efficient.

#### 4.5. Mixed-Mode Examples

To complete this section, a pair of examples are presented using a logic/switch combination and a logic/electrical combination. The circuit to be simulated is a CMOS 4x5 multiplier with a 4-bit counter to generate a test sequence, as shown in block form in Figs. 4.17(a) and 4.17(b). The entire circuit is shown in Fig. 4.18. The multiplier uses the novel adder circuit illustrated in Fig. 4.19(a) and the exclusive-OR circuit of Fig 4.19(b). It is clear from this figure that the adder would be difficult to represent using Boolean gates. Therefore, a switch-level description is appropriate. For the simulation, the multiplier was connected in a multiply-by-2 configuration by setting the B-bits to 2. The counter, shown earlier in conjunction with the CDE cir-



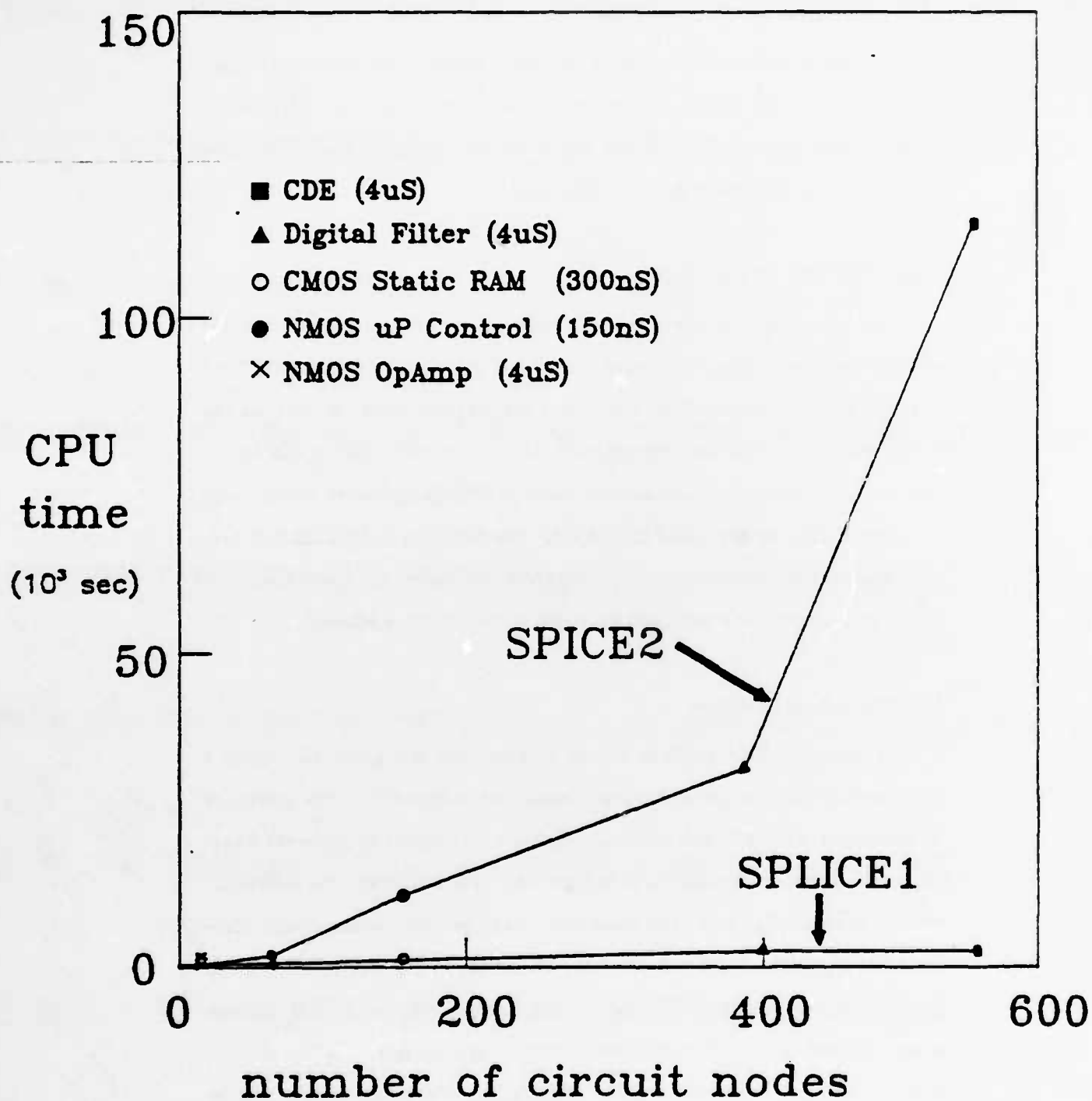


Fig. 4.16 : A plot of the results obtained using SPICE1 and SPICE2.

cuit, was represented at the logic level using boolean gates. It provided a test sequence which was applied to the A-bits.

Initially, the operation of the multiplier was verified at the switch-level. This required only 7.9 CPU-seconds, *including the simulation of the counter circuit*. The output of this simulation is given in Fig. 4.20(a). Note that the outputs, p0, p1, p2 and p3, are evaluated at the input edges, since the simulator is operating in zero-delay mode at the switch-level.

After debugging the circuit at the switch-level, the description of the multiplier was changed from a switch-level description to an electrical-level description by simply changing the underlying models associated with the transistors. The majority of the description was left unchanged. The counter circuit description at the logic level was left in the circuit to generate the test inputs for the A-bits, as before. Logic-to-Voltage converters were inserted where necessary. This simulation required 682.7 seconds. The output of the simulation is shown in Fig. 4.20(b).

Some useful ways to use the mixed-mode capability in SPLICE1 have been illustrated in this example:

- (1) One can debug the circuit very efficiently using zero-delay switch-level simulation. Then, a more detailed simulation can be performed to determine exact delays at the electrical level with very few changes in the circuit description, if the design is described hierarchically.
- (2) A complicated logic circuit can be used to generate test inputs for an electrical simulation as opposed to using logic sources as input. In fact, a master clock signal was the only input waveform for the mixed-mode simulations described here.

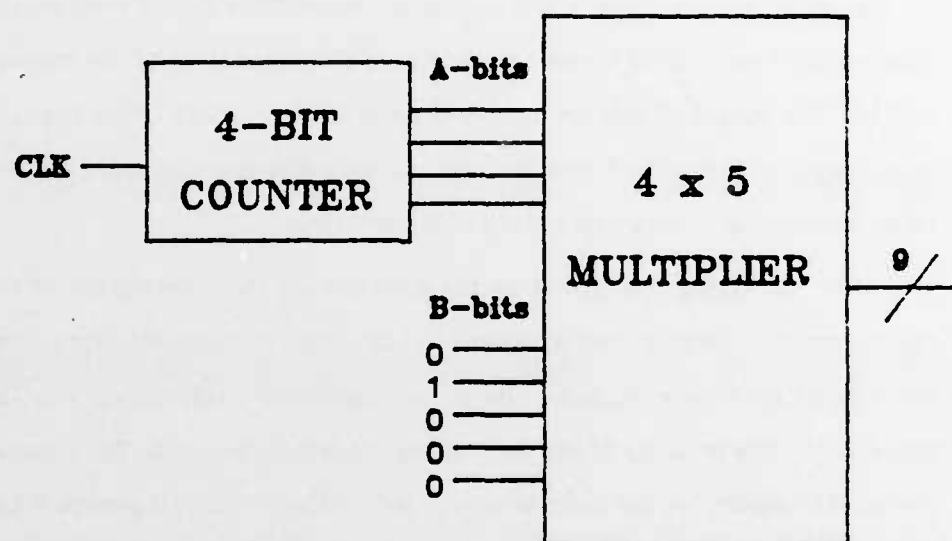


Fig. 4.17 : Block diagram of 4x5 Multiplier Circuit with Counter Circuit

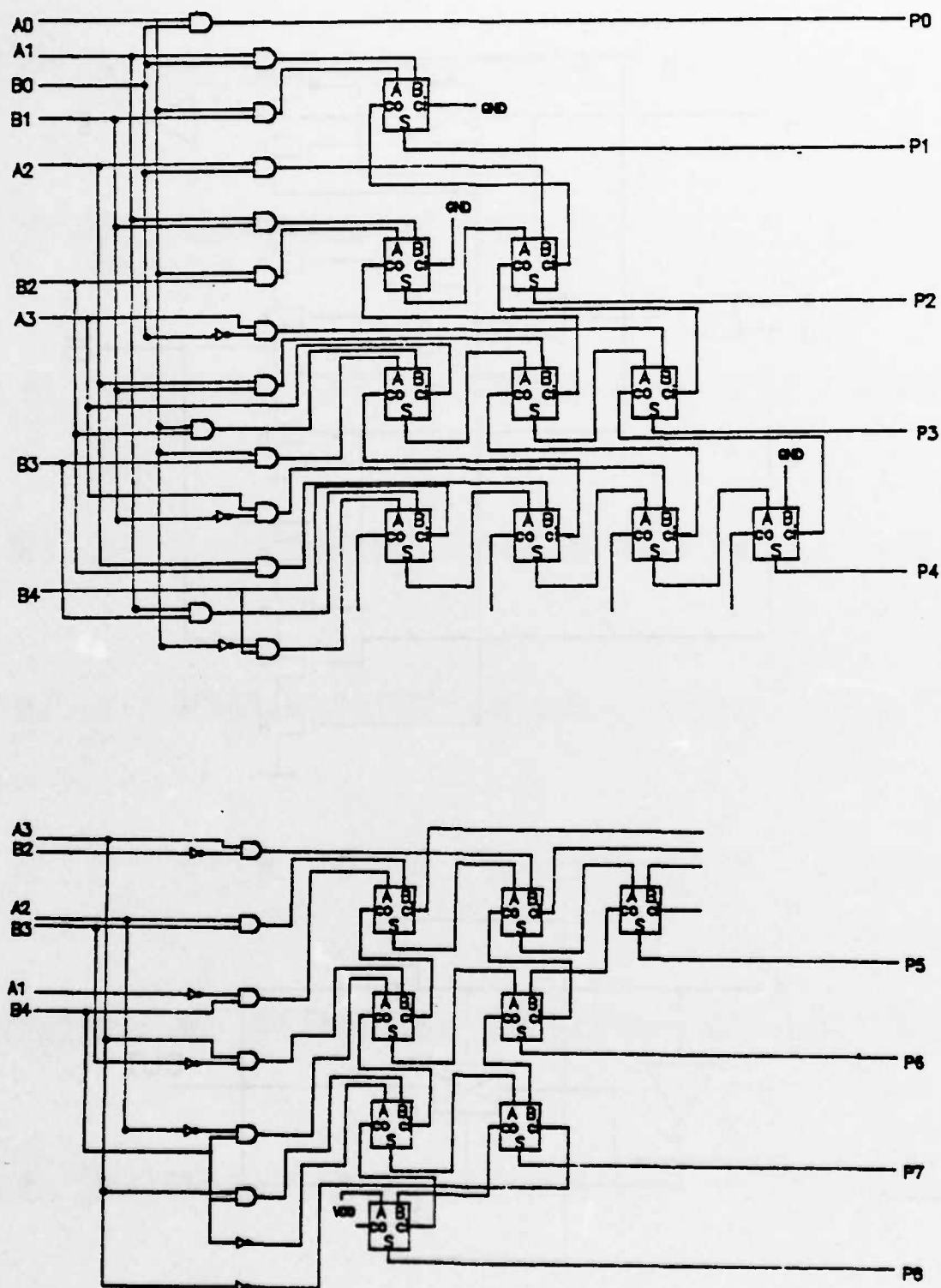


Fig. 4.18 : Details of 4x5 Multiplier circuit

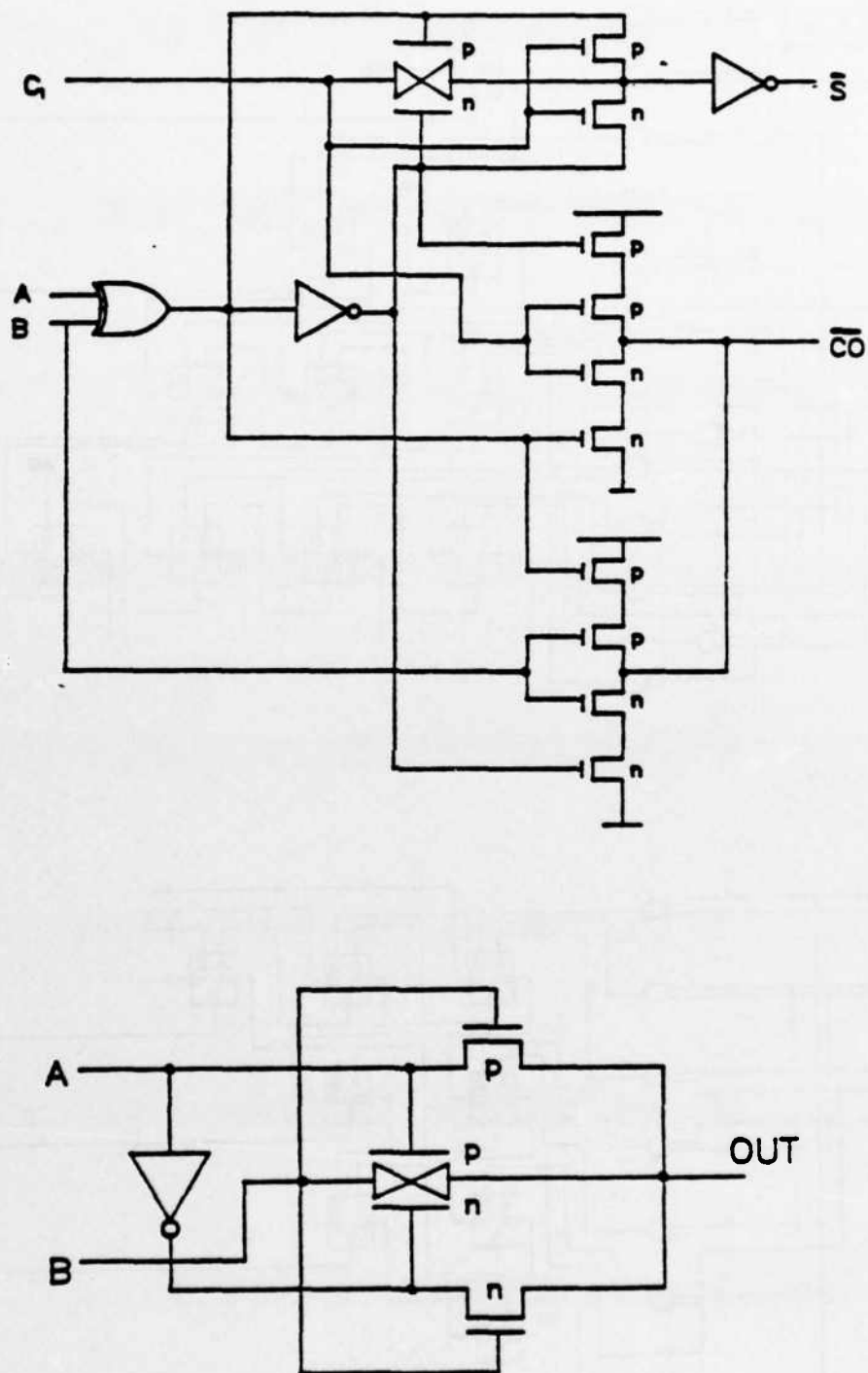


Fig. 4.19 : Details of Adder and Exclusive-OR circuits

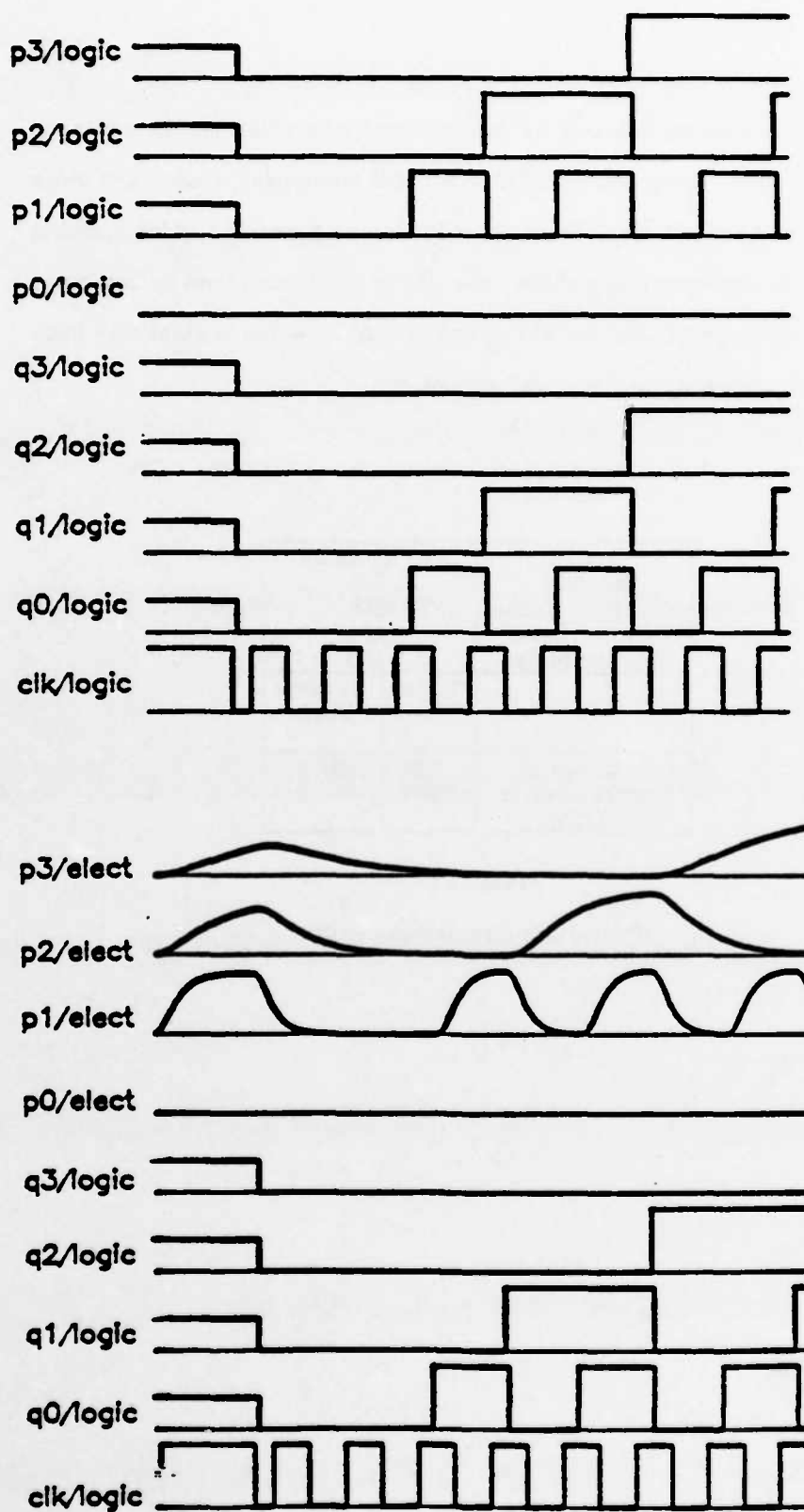


Fig. 4.20 : Output of Switch-level and Electrical-level Simulations

- (3) A complicated circuit can be decomposed into small blocks and each block can be simulated with ITA electrical simulation. Once each block has been checked, a switch-level model can be generated which matches the logic characteristics of the cells. These blocks can then be combined for a switch-level analysis of the entire circuit which is relatively inexpensive compared to electrical simulation.

The results of the simulations are summarized in the table below.

Circuit	4x5 Multiplier	
Mosfets	545	
Multiplier Nodes	248	
Counter Gates	124	
Counter Nodes	130	
	Time (s)	Memory (Kbyte)
Switch-level	7.9	64.5
Electrical-level (ITA)	682.7	68.3

Table 4.5

Mixed-Mode simulation results



## CHAPTER 5

### 5. CONCLUSIONS

SPLICE1 has been greatly improved by incorporating the new techniques described in this report. As evidenced by the statistics in Chap. 4, the new electrical simulation approach, ITA, is substantially faster than SPICE2 and requires far less storage. This method has shown so much promise that efforts are underway to generalize it as a standard circuit simulation approach. As pointed out earlier, the major problem with the method is the number of iterations required to obtain a solution when floating capacitors are present in the circuit. As the prototype program is developed further, it is expected that the performance characteristics will be significantly better than SPICE2. The ITA method provides a way to efficiently simulate large digital circuits and it may replace the standard approach in this application. It is also suitable for implementation on special-purpose hardware and work is underway in this area. Other areas of future work include the extension of the method to use Modified Nodal Analysis, dynamic timestep control and error control mechanisms.

The logic analysis in SPLICE1.7 has been enhanced to perform true-value logic simulation using a strength-oriented MOS model. This not only allows accurate modeling at the logic level but also provides a mechanism to perform accurate mixed-mode simulation. There is still work to be done in the area of strength modeling for logic elements to define the electrical/logic interfaces more accurately. SPLICE1.7 handles logic transfer gates in a consistent manner but the CNR method is not appropriate for a multiprocessor architecture. There is also the issue of delay modeling at the switch-level

which has not been addressed here. Research is currently being directed at applying multiple iterations at the logic level to determine state and delay information in transistor-level logic circuits.

In conclusion, the concepts presented in this report suggest that consistent electrical and logic simulation can be performed at the transistor-level using relaxation-based algorithms and event-driven selective trace techniques.

## References

1. A.R. Newton, "The Simulation of Large-Scale Integrated Circuits," *Memo UCB/ERL M78/52*, University of California, (July 1978). Ph.D. Dissertation.
2. A.R. Newton, "The Simulation of Large-Scale Integrated Circuits," *IEEE Trans. on Circuits and Systems* Vol. CAS-26 pp. 741-749 (September 1979).
3. A.R. Newton, "Timing, Logic and Mixed-mode Simulation for Large MOS Integrated Circuits," pp. 175-240 in *Computer Design Aids for VLSI Circuits*, ed. P. Antognetti, D.O. Pederson, and H. De Man, Sijthoff and Noordhoff (1981).
4. B.R. Chawla, H.K. Gummel, and P. Kozak, "MOTIS - An MOS timing simulator," *IEEE Trans. on Circ. and Sys.* CAS-22 pp. 901-909 (Dec. 1975).
5. S.P. Fan, M.Y. Hsueh, A.R. Newton, and D.O. Pederson, "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," *Proc. IEEE Int. Symp. on Circ. and Sys.*, (April 1977).
6. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *UCB/ERL M75/520*, University of California, Berkeley, (May 1975). Ph.D. Dissertation
7. *LOGIS: User's Manual Version 4*, ISD Corporation (1980).
8. F. Jenkins, *ILOGS: User's Manual*, Simutec (1982).
9. R.E. Bryant, "An Algorithm for MOS Logic Simulation," *LAMBDA*, pp. 46-53 (4th Quarter 1980).
10. C.M. Baker and C. Terman, "Tools for Verifying Integrated Circuit Designs," *LAMBDA* (4th Quarter 1980).

11. J.L. Burns, A.R. Newton, and D.O. Pederson, "Active Device Table Look-up Models For Circuit Simulation," *Proc. 1983 Int. Symp. on Circ and Sys.*, (May 1983).
12. A.R. Newton and A.L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Trans. on Electron Devices*, pp. 1184-1207 (Sept. 1983).
13. "Advanced statistical analysis program (ASTAP)," Pub. No. SH20-1118-0, IBM Corp. Data Proc. Div., White Plains, NY ().
14. N. Tanabe, H. Nakamura, and K. Kawakita, "An MOS Circuit Simulator for LSI," *Proc. IEEE Int. Symp. on Circ. and Sys.*, pp. 1035-1039 (April 1980).
15. G.R. Boyle, "Simulation of Integrated Injection Logic," *ERL Memo. UCB/ERL M78/43*, University of California, (March 1978). Ph.D. Dissertation.
16. J.M. Ortega and W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York (1970).
17. K. Sakallah and S.W. Director, "An Activity-Directed Circuit Simulation Algorithm," *Proc. IEEE Int. Conf. on Circ. and Computers*, (October 1980).
18. E. Cohen, "Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System," *ERL Memo. UCB/ERL M81/29*, (May 1981). Ph.D. Dissertation.
19. A. Vladimirescu and D.O. Pederson, "Performance Limits of the CLASSIE Circuit Simulation Program," *Proceedings of the Int. Symp. on Circ. and Syst.*, (May 1982).

20. E. Lelarasmee, A. Ruheli , and A.L. Sangiovanni Vincentelli, "The Waveform Relaxation Method for the Time-Domain Analysis of Large Scale Integrated Circuits," *IEEE Tran. on CAD of Int. Circ. and Sys.* Vol CAD 1, No. 3 pp. 131-145 (Aug 82).
21. J. White and A. Sangiovanni-Vincentelli, "RELAX2: A New Waveform Relaxation Approach for the Analysis of LSI MOS Circuits," *Proc. 1983 Int. Symp on Circ. and Sys.*, (May 1983).
22. J. E. Kleckner, R. A. Saleh , and A. R. Newton, "Electrical Consistency in Schematic Simulation," *Proc. IEEE Int. Conf. on Circ. and Comp.*, pp. 30-34 (October 1982).
23. R. A. Saleh, J. E. Kleckner, and A. R. Newton, "Iterated Timing Analysis and SPLICE1.6," *Proc. IEEE Int. Conf. on Computer-Aided Design*, (September 1983).
24. L.O. Chua and P.M. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques*, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1970).
25. A.R. Newton, "The Analysis of Floating Capacitors for Timing Simulation," *Proc. 13th Asilomar Conference on Circuits Systems and Computers*, (November 1979).
26. H. Schichman and D.A. Hodges, "Modeling and Simulation of Insulated Gate Field-Effect Transistor Switching Circuits," *IEEE Journ. on Solid State Circuits* Vol. SC-3 pp. 285-289 (Sept. 1968).
27. J.E. Kleckner, *Iterated Timing Analysis and SPLICE2*, To be published
28. G.R. Case, "The SALOGS - A CDC 6600 Program to Simulate Digital Logic Networks," Sandia Laboratory Report No. SAND 74-044 (1975).

29. D. Dumlugol, H. De Man, P. Stevens, and G. Schrooten , "Local Relaxation Algorithms for Event Driven Simulation of MOS Networks Including Assignable Delay Modelling," *IEEE Trans. on CAD of Integrated Circuits*, (July 1983).
30. Graeme Boyle, Private communication.
31. Gregory D. Jordan and Ravi M. Apte, "Modeling of MOS Transistors in a Logic Simulator," *Proc. IEEE Int. Conf. on Circ. and Comp.*, pp. 431-434 (October 1982).
32. C.J. Terman, "Simulation Tools for Digital LSI Design," *Proposal for Ph.D. Research*, Massachusetts Institute of Technology, (December 1981).
33. *The TTL Data Book for Design Engineers - 2nd Edition*, Texas Instrument Incorporated (1976). See flipflop(7474B p.76), counter(74163 p.326), decoder(74154 p.309), encoder(74148 p.291).
34. Jim Kleckner constructed the CDE circuit.
35. D. Senderowicz, "An NMOS Integrated Vector-Locked Loop," *Memo. No. UCB/ERL M82/32*, University of California, (Nov. 1982).
36. S. Kuninobo designed the ADDER circuit.

## APPENDIX I

### Input Files for Example Circuits

The example circuits may be obtained from the University of California at Berkeley



## APPENDIX II

### SPLICE1.6 Data Structures

## APPENDIX II

### SPLICE1.6 Data Structures

- (1) **Nodes:** The node data structure is set up in GENFS for the logic, electrical and vrail nodes.

#### LOGIC NODE :

offset	abbrev.	definition
0	fop	fanout pointer
1	fip	fanin pointer
2	type	=1 (for logic node) =-1 (for logic output node)
3	ts*	fanout schedule time
4	lval	logic value (3-bits for current value b2b1b0 3-bits for previous value b5b4b3 1=0, 2=1, 3=X)
5	lstr	logic strength (16-bits for current value 16-bits for previous value minimum strength = 1 ; maximum strength = 85,536)
6	modptr	1: capacitance at node 2: node decay delay value
7	dectim	node decay time

#### ELECTRICAL NODE :

offset	abbrev.	definition
0	fop	fanout pointer
1	fip	fanin pointer
2	type	= 2 (for electrical node) =-2 (for electrical output node)
3	ts*	fanout schedule time(last time or next time)
4	Vn-1	current node voltage
5	Vn-2	previous node voltage
6	capptrs	points to node capacitance values in rvals
7	tsn-1*	last time processed (associated with Vn-1)
8	tsn-2*	previous time processed (associated with Vn-2)

#### VRAIL NODE :

offset	abbrev.	definition
0	fop	-1 (not used)
1	fip	-1 (not used)
2	type	=5 for a vrail node
3	vn	current node voltage = constant
4	vn-1	previous node voltage = constant = vn

INTEGER information is typically accessed using the nodptr array

i.e.  $\text{info} = \text{imem}(\text{nodptr} + \text{locnod} + \text{ipos})$

$\text{imem}$  : integer memory maintained by memory manager  
 $\text{nodptr}$  : node information data structure origin  
 $\text{locnod}$  : position of 1st piece of info for node  
 $\text{ipos}$  : position of desired info

REAL information is accessed through one more level of indirection:

i.e.  $\text{capacitance} = \text{rmem}(\text{rvals} + \text{imem}(\text{nodptr} + \text{locnod} + 5))$

$\text{rmem}$  : real memory maintained by memory manager  
 $\text{rvals}$  : origin of real value array

- (2) **Fanin and Fanout Lists:** Fanin and fanout lists are stored with the node data structure. Fanins to a node are all elements which can affect the value of the node. Fanouts of a node are all elements which can be affected by a new value at the node. They are set up in the LOGFA, TIMFA and ENDFA sub-routines.

locfol:	0	unused location
	1	element 1 ptr
	2	element 2 ptr
	3	element 3 ptr
		.
		.
		.
	n	- element n ptr

If there is only one element in the fanin list (which is often the case), then this list does not exist. The fl pointer in the node data structure has a -ve sign to denote that it is the element pointer itself.

locfl:	0	scheduler link
	1	element 1 ptr
	2	element 2 ptr
	3	element 3 ptr
		.
		.
		.
	n	- element n ptr

- (3) **Models:** SPLICE1 stores model information using two levels of indirection so that one model may be referenced by many elements.

model info pointers are stored in an array called mdmptr:

mdmptr:	0	locmod 0
	1	locmod 1
	2	locmod 2
	3	locmod 3
		.
		.
		.
	n	locmod n

locmod points into a table called modptr which is organized as follows:

modptr:	0	modtyp 1	(model type)
	1	locpar 1	(location of parameters)
	2	modtyp 2	
	3	locpar 2	
	4	modtyp 3	
	5	locpar 3	
		.	
		.	
		.	

locpar points into rvals which is an array of floating-point quantities and so parameters are accessed as follows:

$$\text{parameter} = \text{rmem} ( \text{rvals} + \text{locpar} )$$

The rvals array is just a set of real values in the rmem space.

rvals :	0	rvalue 0
	1	rvalue 1
	2	rvalue 2
	3	rvalue 3
		.
		.
		.
	n	rvalue n

- (4) **Elements:** Elements are initially written out to scratch files (timel, logel) by the routine SAVEI. Once they are read back in, they are stored in the array elmptr with the following format:

elmptr :	0	-modnum	(first logic element)
	1	noutputs	(number of outputs)
	2	node1	
	3	node2	
	4	node3	
		.	
		.	
	1	-modnum	(second logic element)
i+1		noutputs	(number of outputs)
i+2		node1	
i+3		node2	
		.	
		.	
		.	
nlogwds+0			(last logic element node)
nlogwds+1		-modnum	(first electrical element)
nlogwds+2		noutputs	(number of outputs)
nlogwds+3		node1	
nlogwds+4		node2	
		.	
		.	
		.	
ntimwds+0			(last electrical node)

- (5) **Scheduler:** The time queue is made up of 2 - 100 word arrays and a pool for any events which do not fall within 200 timepoints of the beginning of the queue.

QUEUE 1		time
iscb1 :		0
		1
		2
	.	
	.	
	.	
lscb1 :		99

QUEUE 2		time
iscb2 :		0
		1
		2
	.	
	.	
	.	
lscb2 :		99

POOL	
iscb3 :	TIME 1
	LOCFOL 1
	TIME 2
	LOCFOL 2
	TIME 3
	LOCFOL 3
	.
	.
	.
	-1
lscb3 :	

## APPENDIX III

### SPLICE1.6 Electrical Element Model Equations



## Electrical Element Model Equations

### 1. Resistors

$$G_{eq} = \frac{1}{R}$$

$$I_{eq} = \frac{(V_1 - V_2)}{R}$$

### 2. Floating Capacitors

$$G_{eq} = \frac{C_{float}}{h}$$

$$I_{eq} = C_{float} \frac{(V_1^n - V_1^{n-1}) - (V_2^n - V_2^{n-1})}{h}$$

### 3. Transistors

#### a. Triode Region

$$I_{eq} = \mu C_{ox} \frac{W}{L} (V_{gs} - V_T - \frac{V_{ds}}{2}) V_{ds} (1.0 + \lambda V_{ds})$$

Drain node

$$G_{eq} = \mu C_{ox} \frac{W}{L} ((V_{gs} - V_T - \frac{V_{ds}}{2}) V_{ds} \lambda + (1.0 + \lambda V_{ds}) (V_{gs} - V_T - V_{ds}))$$

Source Node

$$G_{eq} = \mu C_{ox} \frac{W}{L} ((V_{gs} - V_T + V_{ds} \frac{\gamma}{\sqrt{V_{gs} + 2\phi_F}}) (1.0 + \lambda V_{ds}) + (V_{gs} - V_T - \frac{V_{ds}}{2}) \lambda V_{ds})$$

#### b. Saturation Region

$$I_{eq} = \frac{\mu C_{ox}}{2} \frac{W}{L} (1.0 + \lambda V_{ds}) (V_{gs} - V_T)^2$$

Drain Node

$$G_{eq} = \frac{\mu C_{ox}}{2} \frac{W}{L} (V_{gs} - V_T)^2 \lambda$$

Source Node

$$G_{eq} = \frac{\mu C_{ox}}{2} \frac{W}{L} ((V_{gs} - V_T)^2 \lambda + (V_{gs} - V_T) (1.0 + \frac{\gamma}{\sqrt{V_{sb} + 2\phi_F}}) (1.0 + \lambda V_{ds}))$$

## APPENDIX IV

### Source Code for SPLICE1

The SPLICE1 program is available in the public domain from the University of California at Berkeley

# BLOSSOM: An Algorithm and Architecture for the Solution of Large-Scale Linear System

Fu-Hua Ko and  
Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Science  
U.C. Berkeley

## Abstract

Block LU factorization and other partitioned matrix algorithms are used in a VLSI computing system, BLOSSOM, to solve very large-scale matrix problems. With only one type of VLSI arithmetic processing unit needed for submatrix computations, a reconfigurable processor array is used. Since numerical properties are vital to matrix computations, we use neighbor pivoting on submatrices with scalar elements and partial pivoting on full matrix with submatrices as elements. Natural topological structures of certain kinds of operand matrices are also exploited to help reducing pivoting costs. BLOSSOM is designed such that submatrices of different sizes can be efficiently processed even when the number of processing units is less than that of the submatrix elements. Comparisons with other special hardware schemes are provided as a reference.

## 1. Introduction

The solution of Large-scale linear System of algebraic Equations (LSE) is needed in the analysis and simulation of many engineering systems. Finite-element analysis, circuit simulation, and power system analysis are few examples. In these applications, sparse matrix techniques have been used extensively to speed up the solution process. The time complexity of these techniques has been experimentally estimated to be  $O(n^\alpha)$ , where  $1.2 \leq \alpha \leq 1.5$  and  $n$  is the number of equations. When analyzing very large scale systems (more than 10,000 equations), the computational complexity of these techniques makes the solution process very expensive and the use of large main-frames such as the IBM3081 indispensable.

New architectures, in particular vector computers such as the CRAY 1, have inspired the design of new algorithms to exploit parallelism in the solution process. An important example is the program CLASSIE [14] for the simulation of electronic circuits. Along these lines, peripheral array processors, such as the FPS164, can also be used in conjunction with hosts such as the VAX11/780 to speed up the solution process. However, this speedup is not enough to cope with the problems to be solved in the VLSI era.

The advent of VLSI technology has made the cost-effective design of special-purpose machines possible. Examples of these machines are the Yorktown Simulation Engine (YSE) for logic solution [3] and Systolic Arrays [11]. Special purpose machines have also been proposed for the solution of LSE [7, 9, 5, 15]. Most of these machines limit the size of the operand matrix. When no size limit is imposed, the operand matrix has to be partitioned into submatrices of equal sizes. Only Johnson [7] and Pottle [9] treated the related numerical properties and matrix sparsity is exploited only in [9]. However special matrix structures, such as the Bordered Block Diagonal Form (BBDF) or the Bordered Block Triangular Form (BBTF), commonly expected in engineering problem, are not exploited in [9]. In this paper, we propose a new algorithm-architecture BLOSSOM for the solution of LSE.

The paper is organized as follows. In Section 2 we propose a variant of block LU decomposition with block neighbor pivoting to ensure numerical stability and accuracy. A parallel-pipeline architecture, described in Section 3, is designed to implement the block LU decomposition. This architecture supports other matrix operations used as sub-procedures by block LU decomposition such as the multiplication and the inversion of submatrices. In Section 4, we describe the hardware implementation of these matrix operations. Finally, a comparison with other hardware schemes is listed in table 1.

## 2. Block LU Factorization

$$A = LU$$

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} = \begin{bmatrix} L_{11} & & & 0 \\ & L_{22} & & \\ & & \ddots & \\ & & & L_{nn} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & \dots & U_{1n} \\ & U_{22} & \dots & U_{2n} \\ & & \ddots & \\ 0 & & & U_{nn} \end{bmatrix}$$

Fig. 1 Block LU decomposition

Let  $A \in \mathbb{R}^{N \times N}$  be partitioned into  $n^2$  submatrices as shown in Fig. 1, where  $1 \leq i \leq n$  and  $\sum_{i=1}^n m_i = N$ . Note  $A_{ij} \in \mathbb{R}^{m_i \times m_j}$ .

Let  $L_{ij}$  and  $U_{ij}$  denote respectively the submatrices of the block LU factors of  $A$  in the  $(i, j)$  and  $(j, i)$  positions. The following algorithm performs block LU decomposition:

### Algorithm 1.1. Block LU-decomposition:

- (1)  $i = 1$ ;
- (2) Compute  $L_{ii} = A_{ii} - \sum_{k=1}^{i-1} L_{ik} U_{ki}$ ;
- (3) Compute the inverse of  $L_{ii}$ , denoted by  $L_{ii}^{-1}$ ;
- (4) If  $i = n$ , stop.
- (5) Compute for all  $k, i < k \leq n$ :  

$$L_{ki} = A_{ki} - \sum_{j=1}^{i-1} L_{kj} U_{ji}$$

$$U_{ik} = L_{ii}^{-1} (A_{ik} - \sum_{j=1}^{i-1} L_{ij} U_{jk})$$
- (6)  $i = i + 1$ ; go to (2);

Note that the structure of this algorithm is the same as Crout's method except that each step is based on submatrices instead of scalar elements. Thus, the block  $L, U$  factors are not triangular matrices but rather block triangular matrices and the multipliers  $L_{ii}$  in step 3 are computed by means of a matrix inversion instead of scalar inversion.

For the algorithm to be well-defined, it is necessary that  $L_{ii}$  be nonsingular. George [6] computed the block LU factorization under the condition that the operand matrix is diagonally dominant. However, Bunch [1] pointed out that diagonal dominance of  $A$  is not sufficient to guarantee that the algorithm is well defined. Bunch conjectured that

Block diagonal dominance\* is a sufficient condition. The following theorem states that strictly block diagonal dominance guarantees the nonsingularity of the diagonal submatrices of each reduced matrix of  $A$ , which implies the existence of block LU factors. The proof can be found in [10].

**Existence Theorem:** Let  $A$  be a nonsingular  $N \times N$  matrix partitioned as in Fig. 1. If  $A$  is strictly block row diagonally dominant, then each reduced matrix of  $A$  in block Gaussian elimination is strictly block (row) diagonally dominant.

**Remark:** The uniqueness of block LU factors is proved in [10].

Sparsity has been exploited extensively in scalar LSE solution algorithms to reduce computational complexity. This concept is used block-wise in BLOSSOM. Operations performed by BLOSSOM involve nonzero blocks only. To exploit the sparsity further, concurrent subtasks of block LU decomposition on matrices of special topological structures such as BBDF, BBTF can be executed in parallel by BLOSSOM. These subtasks are, for example, inversion of diagonal submatrices on the block diagonal of a matrix in BBDF, forming of  $L$  factors on the  $i$ 'th block column, etc.

Dahlquist et al[2] indicated that a compact scalar Gaussian elimination algorithm can not incorporate complete or diagonal pivoting easily because of the order in which LU factors are computed. This limitation also applies to block LU decomposition. When partial pivoting is used in block LU decomposition, square blocks in the same block column of the present pivot are checked to find the one with smallest-norm inverse. We add partial pivoting to algorithm 1.1 as follows: after  $L_{ik}$ 's are computed (including  $L_{kk}$ ), compute their inverses. Compare the norms of these inverses to determine a new pivot, and then compute  $U_{ik}$ 's. The hardware implementation of block LU decomposition is thus based on algorithm 1.1 with partial pivoting added.

### 3. System Architecture

The BLOSSOM system consists of a host-interface, an executive control unit (ECU), a sequencer, and a reconfigurable processor array as depicted in Fig. 2. The following is a behavioral description of each functional block.

#### 3.1. Host-interface

The host machine acts as an intelligent interface between the user and BLOSSOM. A request for a LSE solution issued by the user is sent to BLOSSOM by the host machine. If BLOSSOM is not being occupied, data will be loaded into BLOSSOM, freeing the host for other duties.

BLOSSOM is intended to be capable of being attached to different kinds of host. For this reason, the host-interface consists of two parts. The first part is dedicated to accommodate different host machines by providing facilities such as converting integer numbers into floating point numbers and matching different word lengths. The second part initiates block LU decomposition as described in the sequel.

The data path and control path are generally kept separate in most processor designs. In BLOSSOM, the operation set is very small and the amount of data is very large. Hence control signals are embedded into the data stream. Since the host-interface recognizes only partitioned matrices and vectors, the host provides a data separator for each submatrix and vector segment in the data stream. The host-interface generates a proper data representation each time a separator is encountered. This data representation and the instruction words are then sent to ECU, while data

\*  $A$  is block (row) diagonally dominant if  $A_{ii}$  is nonsingular and  $\sum_{j \neq i} \|A_{ij}\| \leq \|A_{ii}\|$  for  $i=1, \dots, n$ , where  $\|\cdot\|$  denotes an operator norm.  $A$  is strictly block (row) diagonally dominant if the strict inequality holds for each  $i$ . [4]

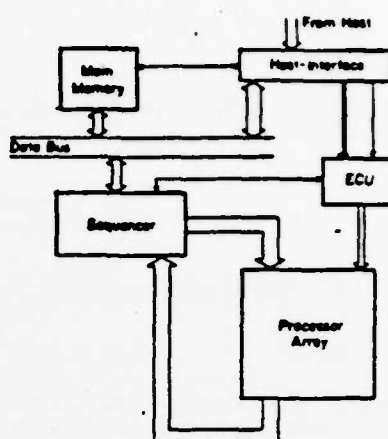


Fig. 2 Block Diagram of System Architecture of BLOSSOM are sent to the main memory.

#### 3.2. Executive Control Unit (ECU)

The ECU decodes host-interface instructions on matrices into sequencer instructions on submatrices and vector segments. Thus each task requested by the host is partitioned into several subtasks and each subtask is carried out by the processor array under supervision of the sequencer.

The set of sequencer instructions corresponding to each host instruction is stored in a control memory of ECU, and the status of the sequencer is monitored by the ECU. The ECU also generates control signals indicating whether processors in adjacent rows (columns) should be connected. We denote them by  $RECONr.i$  ( $RECONc.j$ ).

The subtasks under execution by the sequencer are monitored by the ECU. Whenever a subtask is completed, the ECU can perform one of the following actions on the next subtask on the queue: execute it immediately, wait until more processors become available, or wait until all processors become available. The choice among these possible actions is determined by the sizes of the subtask and the available processor subarrays.

#### 3.3. Sequencer

The ECU generates concurrent requests for the sequencer, which in turn initiates and monitors actions in the processor array. The control logic in the sequencer is then divided into four parts: the Task Control Unit (TSU), the Memory Management Unit (MMU), the Sequencing Unit (SU), and the Feedback Buffer (FB), as shown in Fig. 3.

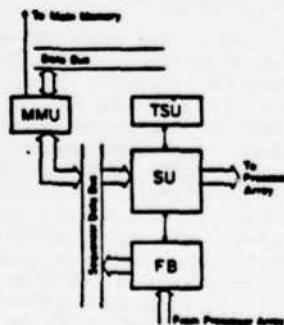


Fig. 3 Block Diagram of Sequencer

### 3.3.1. Task Control Unit & Memory Management Unit

The TSU interfaces with the ECU and maintains the status registers. Each instruction received from the ECU is directly added (instead of decoded) into the data stream that is loaded by the MMU and sequenced by the SU.

The MMU manages the main memory system. Since the amount of data of each operand matrix can be very large, virtual addressing scheme is used. The operand address received from the ECU is the starting virtual address of a submatrix or vector segment. The MMU computes the ending virtual address from it and translates both virtual addresses to physical addresses. The MMU is required to fetch data for concurrent tasks from the multiport memory. The host-interface would try to allocate potentially parallel data objects into different parts of the memory hierarchy to be fetched through different ports. However, memory conflicts are possible when pivoting is employed. If conflicts happen, the MMU would inform the TSU. The latter can either reject the subtask and return it to ECU, or keep the subtask in its queue till the conflict disappears.

### 3.3.2. Sequencing Unit & Feedback Buffer

The SU adds different delays to data loaded by the MMU before sending them to different processor rows or columns according to the requirements of the computing algorithms. The computation of the number of these delays is done in the SU under the control of the TSU. Since the delay is uniform from processor column to column and from processor row to row in each subtask, only one computation is needed for each subtask. The computations for different subtasks are done sequentially to limit the hardware complexity.

The FB is used when the data output from the processor array is resent to the processor array again. To the SU, the FB is a multiport buffer between it and the processor array. The FB has the capability to reverse the existing delay relationship among a row of data.

### 3.4. Processor Array

The processor array consists of processor elements, local data links, local control links, SU data registers, FB data registers, and the ECU RECON buses, as shown in Fig. 4. The local data links carry multiplexed data words. There is one ECU RECON bus per processor row and one per processor column.

Each subprocedure used by block LU decomposition has a corresponding microprogram stored in each processor, which is activated when that subprocedure is initiated by the sequencer. These subprocedures assume that all segments of a row (or column) of an operand are stored in one processor row (or column). A register file in each processor is used as a queue to enable such a storage scheme. Each processor has four data buses connecting to its neighbors which are named "data ports". The processors also make individual decisions on whether connections to neighboring processors are maintained or not by invoking proper microprograms.

### 4. Partitioned Matrix Operations

The procedures employed by block LU decomposition may use any or all forms of matrix, submatrix, vector, vector segment, and scalar for their operands. A procedure is implemented by a string of commands issued by its controlling module, and is labeled by its controlling module. For example, block LU decomposition is ECU controlled. Due to the present space limitation in this paper, only submatrix inversion is described in details here. Other operations are listed for reference only.

#### 4.1. Pipelined Inversion of a Submatrix

Let  $PEP^T$  be the operand matrix and  $IERP^T$  be the

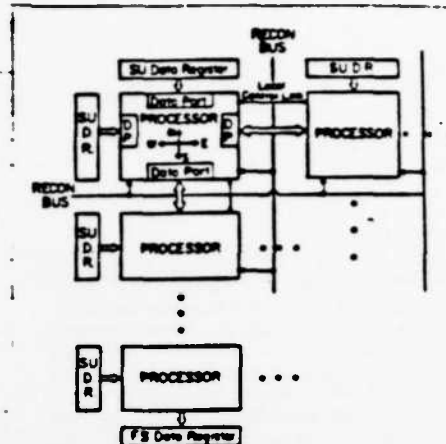


Fig. 4 Processor Array

unity matrix. The pipelined inversion is essentially Gaussian elimination applied to the augmented matrix  $[P, I]$  resulting in  $[I, P^{-1}]$ . The procedure is listed below:

```

for j=1 to p {
   $B_{jj} = \frac{1}{P_{jj}}$ ;
   $B_{js} = B_{js} B_{jj}$ , for all s=1 to p;
  if i ≠ j
    then  $P_{ik} = P_{ik} - P_{ij} P_{jk}$ , for all k, i=1 to p;
     $I_{ik} = I_{ik} - P_{ij} I_{jk}$ , for all k, i=1 to p;
}

```

Note that  $B_{ij} = [P_{ij} | I_{ij}]$ , and  $P_{ij}$ ,  $I_{ij}$  are scalars.

Two important features are added to the hardware implementation of this procedure: "folding of operand submatrix" and "neighbor pivoting". The inverting process is partitioned into two phases. The first phase is triangularization with neighbor pivoting. Backward substitution is the second. The two phases are executed as two contiguous steps following a one-time loading of data.

When neighbor pivoting is used, the leading element of the next adjacent row is reduced to zero by adding a multiple of the current row. When this multiplying factor exceeds unity, the former row becomes the current row and the latter has its leading element reduced to zero. Thus this factor is kept fractional, which suggests that this triangularization process is numerically stable as confirmed by empirical results [15, 13].

Let  $P_j < k >$  denote the k'th segment of the j'th row of "P" and  $P_j < k > g$  denote the g'th nonzero element of  $P_j < k >$ . The same notation applies to the unity matrix "I" and every operation applied to  $P_j$  is applied to  $I_j$  as well. We describe next the implementation of the algorithm on a per-processor-row basis. The following symbols are used:  $P_i$  denotes the current pivot row on that particular processor row,  $P_i$  the row that is currently processed,  $P_0$  the row that has its leading element reduced to zero and sent to the next processor row, "pr" the processor row index, "ino" the iteration number of the whole operand in the processor array, "q x g" the size of the processor array, and "h" the number of segments for any one column (or row) of P after folding.

#### Algorithm 4.2 1 Triangularization:

- (1) ino = 1;
- (2) If  $P_i < ino > 1 < P_i < ino > 1$  then go to (6);
- (3)  $m = \frac{P_{ik} < ino > 1}{P_i < ino > 1}$ ;
- (4)  $P_0 = P_i - m P_i$ ;



- (5) goto (9);
- (6)  $m = \begin{cases} p - i_{no} & \text{if } p < i_{no} \\ 1 & \text{otherwise} \end{cases}$ ;
- (7)  $P_r = P_r - m P_i$ ;
- (8)  $P_r = P_i$ ;
- (9)  $i_{pi} = i_{pi} + 1$ ;
- (10) If  $i_{pi} < (p - q + i_{no} - 1)$  then go to (14);
- (11)  $i_{pi} = 1$ ;  $P_r = P_i$ ;
- (12)  $i_{no} = i_{no} + 1$ ;
- (13)  $n_r = p - q + i_{no} - 1$ ;  $P_{ri} = P_r$ ;
- (14) If  $i_{no} \leq h$  then go to (2);
- (15) stop;

In this algorithm, each element of a segment of a row of "P" is processed in a way that it lags or leads its neighbor-column elements by one step as in most algorithms used in systolic arrays. The multiplier "m" is generated from the diagonal position and transmitted to every column. Since the "I" matrix has different zero-nonzero topological structure from the "P" matrix, the algorithm has two distinct phases for "P" and "I". The "I" phase can significantly lag the progress of "P" matrix in time, but the operations are essentially the same. A similar backward substitution algorithm can be found in [10].

#### 4.2. Other Matrix Operations

The operation set of BLOSSOM also includes: system solution and matrix multiplication (both controlled by the ECU), submatrix multiplication, submatrix addition, submatrix norm computation, and submatrix-vector multiplication (all controlled by the SU). Detailed description of these operations are listed in [10].

#### 5. Concluding Remarks

BLOSSOM differs from other special purpose machines by using block LU decomposition and by exploiting matrix sparsity commonly expected in engineering problems. Table 1 lists the comparison of BLOSSOM with other hardware schemes proposed in [12, 7, 9, 8, 5], where the complexity of  $O(Nwh^2)$  for a LSE solution by BLOSSOM happens in the case of  $N \times N$  BBDF matrices under mild conditions [10]. With the architecture designed to exploit the parallelism of the block LU decomposition algorithm, BLOSSOM is efficient in solving problems encountered in the VLSI era. Current work on BLOSSOM also involves its simulation, its performance evaluation, and the expansion of its operation set.

#### 6. Acknowledgement

The authors wish to thank Mr. Giovanni De Micheli for the helpful editorial comment. This work was sponsored in part by the Naval Electronic Systems Command contract (DARPA N00039-83-C-0107).

#### References

1. James R. Bunch, "Block Methods for Solving sparse Linear Systems," in *Proceedings of the Symposium on Sparse Matrix Computations*, September 1975.
2. Germund Dahlquist, Ake Björck, and Ned Anderson, *Numerical Methods*, Prentice-Hall, 1974.
3. M.M. Denneau, "The Yorktown Simulation Engine," in *Proceeding of the 19th Design Automation Conference*, pp. 55-59, 1982.
4. David G. Feingold and Richard S. Varga, "Block diagonally dominant matrices and generalizations of the Gershgorin circle theorem," *Pacific J. of Math.*, pp. 1241-1250, 12(1962).
5. Ron J. Gal-Ezer, K. S. Arun, and D.V. Bhaskar Rao, and Sun-Yuan Kung, "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Trans. on Computers*, vol. C-31, no. 11, November 1982.

6. Alan George, "On block elimination for sparse linear systems," *SIAM Numerical Analysis*, pp. 585-603, 1974.
7. Lennart Johnsson, *A Computational Array for the QR Method*, MIT Conference in Advanced Research in VLSI, January 1982.
8. Kai Hwang and Yeng-Heng Cheng, "Partitioned Matrix Algorithms for VLSI Arithmetic Systems," *IEEE Transactions on Computers*, vol. C-31, December 1982.
9. R.M. Kieckhafer and Christopher Pottle, "A Processor Array for Factorization of Unstructured Sparse Matrices," in *ICCC Confer. Proc.*, 1982.
10. Fu-Hwa Ko and Alberto Sangiovanni-Vicentelli, *BLOSSOM: An Algorithm and Architecture for the Solution of Large-scale Linear Systems*, Initial Report.
11. H.T. Kung, "Why Systolic Architectures?," *IEEE Computer*, January 1982.
12. H.T. Kung and C.E. Leiserson, "Algorithms for VLSI Processor Arrays," in *Symposium on Sparse Matrix Computations and Their Applications*, 1978.
13. L.D. Rogers, *Optimal Paging Strategies and Stability Considerations for Solving Large Linear Systems*, 1973. Ph.D. thesis, University of Waterloo, Department of Computer Science.
14. Andrei Vladimirescu, *LSI Circuit Simulation on Vector Computers*, October 1982. Memo. No. UCB/ERL M92/75.
15. W.M. Gentleman and H.T. Kung, "Matrix Triangularization by Systolic Arrays," in *Proc. SPIE Symp.*, vol. 298, SPIE 1981.

Table 1. Comparison of hardware schemes on LSE solution

Attributes	Systolic Array by H.T. Kung et al	Operational Array for QR by Johnsson	LU on Orthogonal sparse matrix by Pottle	Partitioned matrix algo by Hwang and Cheng	BLOSSOM	WAP by S.Y. Kung et al
LSE Soln. Algorithm	Needs FE & BS facilities	Needs BS facility	Needs FE and extra BS facilities	Needs FE and extra BS facilities	Offered	Offered
Size limit	Window size $\leq$ operand size (PA)*	sz(PA)	Computational window $\leq$ sz(PA)	No limit if segmented	No limit for any size	Not for all sizes unless expanded
Pivoting Strategy	no	NeP	no need	None on the fly	None	NeP and ParP**
Inter-processor Comm.	Local Sync.	Pipeline	Local Pipeline. Some broadcast. Sync.	Not specified	Local Pipeline	Handshake mechanism
Complexity	$3n + \text{run}(r,s)$	$3n$	$m + 3n + \text{sz}$	$n^2$	$O(N^2)$ or $O(\frac{N^2}{m})$ or $O(\frac{N^2}{m} + 1)$	$n$
Number of processors	$m$	$\frac{n^2 + n}{2}$	$r(3r + s) + 2$	$\text{sz}_{\max}^2$	$O(\sqrt{N})$ or $O(N/m)$	$n^2$
Feasible operand prop.	Has L, U factors	Any matrix within size limit	Any matrix within size limit	Needs to Pivot: be sym-block in each factorized LU	Block diagonal dominance	Same as Hwang's if extended

\* sz(PA) denotes size of processor array (n x n in most cases)

\*\* NeP denotes neighbor pivoting. ParP denotes partial pivoting

$r + s - 1$  is the bandwidth of a banded matrix

$\bar{r}$  denotes the mean degree of columns of a sparse matrix

$N$  denotes size of operand matrix.  $m$  denotes submatrix size

(1) matrix is full (2)  $w$  is the original border width.  $A$  is a quantity related to  $x$  and the sizes of operand submatrices

$\&_{\max}$  denotes the maximum number of nonzeros in any row of the operand



## A MULTIPROCESSOR IMPLEMENTATION OF RELAXATION-BASED ELECTRICAL CIRCUIT SIMULATION

J. T. Deutsch and A. R. Newton  
Department of Electrical Engineering  
and Computer Sciences  
University of California, Berkeley, 94720.

**Abstract:** The electrical circuit simulation of large integrated circuits is very expensive. New relaxation-based algorithms promise to reduce this cost by exploiting the properties of large networks. However, this speed improvement is not sufficient for the cost-effective analysis of very large circuits. While array processors have helped improve the performance of circuit simulators, further improvement can be achieved by the use of special-purpose multiprocessors. In this paper, the implementation of a relaxation-based circuit simulation algorithm, called Iterated Timing Analysis, on a multi-processor is described. Initial results indicate that this approach has a great deal of potential for reducing the cost of circuit simulation.

### 1. INTRODUCTION

The use of modern CAD tools, in particular Connectivity Verification Systems (CVS), in the design of complex integrated circuits has increased the probability that circuits will work on first silicon[1]. However, the time lag between a functional circuit and a circuit that meets its performance objectives is increasing. Simple delay estimation techniques and cell-level circuit simulation can be used for first-order performance estimation in constrained design methods. Unfortunately, these approaches do not predict circuit performance accurately for state-of-the-art circuit designs. For this reason, circuit simulators, originally designed to simulate circuits containing under 100 transistors, are often used today to simulate circuits containing many thousands of transistors.

One of the most common analyses performed by circuit simulators and the most expensive in terms of computer time is nonlinear, time-domain transient analysis. By performing this analysis, precise electrical waveform information can be obtained if the device models and parasitics of the circuit are characterized accurately. Because of the need to verify the performance of larger circuits, many users have successfully simulated circuits containing thousands of transistors despite the cost. For example, a 700 MOSFET circuit, analyzed for 4 $\mu$ s of simulated time with an average 2ns time step, takes approximately 4 CPU hours on a VAX 11/780 VMS computer with floating-point accelerator hardware using the SPICE2 program[2].

Gate-level logic simulators (e.g. [3,4]) and switch-level simulators[5-7] can verify circuit function and provide first-order timing information more than three orders of magnitude faster than a detailed circuit simulator. However, to verify circuit performance for critical paths, memory design, and analog circuit blocks, and to detect dc circuit problems such as noise margin errors or incorrect logic thresholds, it is often essential to perform accurate electrical simulation. In some companies the simulation of circuits containing many thousands of devices is performed rou-

tinely and at great expense. In recent years, considerable effort has been focussed on techniques for improving the speed of time-domain electrical analysis while maintaining acceptable waveform accuracy.

A number of approaches have been used to improve the performance of conventional circuit simulators for the analysis of large circuits. The time required to evaluate complex device model equations has been reduced using table-lookup models[8,9]. Techniques based on special-purpose microcode have been investigated for reducing the time required to solve sparse linear systems arising from the linearization of the circuit equations[10]. Node tearing techniques have also been used to exploit circuit regularity by bypassing the solution of subcircuits whose state is not changing[11,12].

These techniques, and others, have also been used to exploit the vector processing capabilities of high performance computers such as the CRAY-1[13,14] and FPS-164[15]. These special-purpose computers have additional hardware designed to exploit the parallelism and pipelining that is available in the programs they execute. Unfortunately, circuit simulation programs are not well suited to these computers. In particular, the sparsity of the circuit matrix and its irregular structure cause the data gather-scatter time to dominate overall program execution time[14]. That is, simply fetching the data stored in memory and writing it back out again after it has been processed becomes the bottleneck. In all cases, the overall speed improvement of the simulation has been at most an order of magnitude, for practical circuits.

Recently, a new class of algorithms has been applied to the electrical IC simulation problem. New simulators using these methods provide *guaranteed* accuracy[16] - as accurate or more accurate waveforms than standard circuit simulators with up to two orders of magnitude speed improvement for large circuits[17,18]. These simulators have been used for the analysis of both digital and analog MOS ICs. They use *relaxation* methods for the solution of the set of ordinary differential equations, (ODEs) which describe the circuit under analysis, rather than the direct, sparse-matrix methods on which standard circuit simulators are based. While these new algorithms provide substantial speed improvements on conventional computers, they can provide much greater speedups on special-purpose hardware that is designed to exploit the particular features of these algorithms[22].

In this paper, the use of the *Iterated Timing Analysis*[18-20] (ITA) on a special-purpose multi-processor is presented. The ITA method is an SOR-Newton, relaxation-based method which uses event-driven analysis and selective trace to exploit the temporal sparsity of the electrical network[21]. Because event-driven selective trace techniques are employed, this algorithm lends itself to implementation on a data-driven computer. Initial results indicate that data-driven multiprocessors, working with a conventional host, can provide almost limitless performance improvement for electrical circuit simulation. This particular class of machines is also well-suited to other network-

graph-based, event-driven algorithms, including fault simulation, layout compaction, layout-rule checking, and the IC tape-out process for fabrication. Many non-electrical problems also fit this model.

A prototype simulator has been implemented on an existing multiprocessor and experimental results from this simulator are presented.

Throughout the paper, a common circuit example is used. It is an industrial NMOS digital filter circuit whose netlist and parasitic capacitor values were obtained from an analysis of the mask layout. The circuit contains 398

electrical nodes and 698 MOSFETS, of which approximately 31% are depletion loads, 42% are driver devices (either source or drain node connected to ground), and the remaining 27% are transfer gates (all three terminals are connected to signal nodes or clocks).

In Section 2, the ITA algorithm is reviewed. In Section 3, the implementation of ITA on a multiprocessor is described and the MSPLICE program is introduced. Section 3 also includes the results of early simulations of the interconnection network for this problem and some extrapolations to more advanced machines. The prototype multiprocessor used to gather experimental data is described in Section 4 and the data obtained from the experimental implementation is presented.

## 2. ITERATED TIMING ANALYSIS

The ITA method is a new form of electrical analysis which can be derived from timing simulation[8]. This form of relaxation-based electrical analysis has shown promising results over a wide class of circuits, from large digital circuits to complex analog designs[18]. The SPLICE1 program, which employs ITA for electrical analysis, is now being used successfully at a number of industrial sites.

The starting point for a description of ITA is the electrical circuit equation formulation. A Nodal Analysis[23] formulation will be used to illustrate the ITA algorithms. Under the assumptions:

- All resistive elements, including active devices, are characterized by constitutive equations where voltages are the controlling variables and currents are the controlled variables.
- All energy storage elements are two-terminal, possibly nonlinear, voltage-controlled capacitors.
- All independent voltage sources have one terminal connected to ground or can be transformed into independent current sources with the use of the Norton transformation.

the nodal network equations, where there are  $N$  equations in  $N$  unknown node voltages,  $N+1$  nodes in the circuit, and node  $N+1$  is the reference node, or ground, can be written:

$$C(v,u)\dot{v} = -f(v,u) \quad (2.1)$$

$$v(0) = V.$$

where  $v(t) \in \mathbb{R}^N$  is the vector of node voltages at time  $t$ ,  $\dot{v}(t) \in \mathbb{R}^N$  is the vector of time derivatives of  $v(t)$ ,  $u(t) \in \mathbb{R}^M$  is the input vector at time  $t$ ,  $C(\cdot): \mathbb{R}^N \times \mathbb{R}^M \rightarrow \mathbb{R}^{N \times N}$  represents the nodal capacitance matrix,  $f: \mathbb{R}^N \times \mathbb{R}^M \rightarrow \mathbb{R}^N$ , and:

$$f(v(t), u(t)) = [f_1(v(t), u(t)), \dots, f_N(v(t), u(t))]^T$$

where  $f_i(v(t), u(t))$  is the sum of the currents charging the capacitors connected to node  $i$ . The differential equations are converted to a set of nonlinear, algebraic difference equations using a stiffly-stable integration formula to give:

$$g(x) = 0 \quad (2.2)$$

where  $x \in \mathbb{R}^N$  is the vector of node voltages at time  $t_{n+1}$ .

and an iterative relaxation method (Gauss-Jacobi or Gauss-Seidel on a uniprocessor) is then used to solve them. However, unlike timing analysis where a single relaxation iteration is used per time-point, in the ITA approach the relaxation process is continued to convergence at a time-point.

Only one Newton-Raphson iteration is used to approximate the solution of each nodal equation per relaxation iteration and event-driven, selective trace techniques may still be used to exploit latency, as for timing simulation. Since in ITA the nonlinear circuit equations are solved by an iterative method until satisfactory convergence is achieved, the numerical properties of the integration methods used to discretize the circuit equations are retained. Thus, the stability and the accuracy problems typical of the timing simulation algorithms are not an issue here[18].

The following algorithm, written in "Pidgin C"[24], illustrates the principle steps involved in ITA analysis, using a Gauss-Seidel iteration, for use on a conventional computer. At each time at which one or more nodes are scheduled to be processed, two event lists,  $E_n(t_n)$  and  $E_{n+1}(t_n)$  are used to separate the nodes to be processed in successive iterations,  $k$  and  $k+1$ , of the Gauss-Seidel-Newton process.

*Gauss-Seidel-Newton Iteration:*

put all nodes that are connected to independent sources in event list  $E_n(0)$ ;

$t_n = 0$ ;

while ( $t_n < TSTOP$ ) {

$k \leftarrow 0$ ;

while (event list  $E_n(t_n)$  is not empty) {

foreach ( $i \in E_n(t_n)$ ) {

$$v_i^{k+1} = v_i^k - \frac{g_i(\tilde{v}^{k+1,k})}{g_i'(v_i^{k+1,k})}$$

$$\text{Where } \tilde{v}^{k+1,k} = [v_1^{k+1}, \dots, v_i^{k+1}, v_{i+1}^k, \dots, v_N^k]^T$$

if ( $|v_i^{k+1} - v_i^k| \leq \epsilon$ ; i.e. convergence is achieved) {

use ITE to determine the next time,  $t_{n+1}$ , for processing node  $i$ ;

add node  $i$  to event list  $E_{n+1}(t_{n+1})$ ;

}

add node  $i$  to event list  $E_{n+1}(t_n)$ ;

add the fanout nodes of node  $i$  to event list  $E_n(t_n)$  if they are not already on  $E_n(t_n)$ ;

}

$E_n(t_n) \leftarrow E_{n+1}(t_n)$ ;  $E_{n+1}(t_n) \leftarrow \text{empty}$ ;

$k \leftarrow k+1$ ;

}

$t_n \leftarrow t_{n+1}$ ;

}

where  $t_n$  is the present time for processing and  $t_{n+1}$  is the next time in the time queue at which an event was scheduled. In this way, the "time-step" is handled independently for each node. The foreach construct requires that the block be executed for each member of the set in a specified order.

This simplified algorithm does not illustrate how such issues as time-step reduction and local truncation-error estimation are handled. These and other important details of the algorithm are described elsewhere[20]. While a nodal formulation was used to describe the approach, a Modified Nodal formulation[25] can also be derived.

As mentioned earlier, the ITA method has guaranteed convergence properties. However, for tightly-coupled portions of a large circuit (e.g. active MOS transmission gate trees or an Operational Amplifier in unity gain configuration) the convergence rate may be relatively slow. For these sub-circuits, a direct approach will usually improve the overall performance of the simulation. A modification of the above algorithm which replaces the single circuit node with an entire subcircuit, described by a nodal admittance matrix, has been used successfully in relaxation-based simulators to improve convergence rate for circuits of this type[21][17]. Applying ITA to subcircuits rather than single electrical nodes has important implications in the multi-processor case. In particular, it permits a tradeoff between the amount of time spent on any single processor at an iteration and the time spent communicating the results of the analysis; a key requirement for deriving maximum efficiency from the multiprocessor, as will be seen later.

### 3. IMPLEMENTATION OF ITA ON A MULTIPROCESSOR

#### 3.1 Introduction

For the purpose of this description, a multiprocessor will be defined as a collection of  $N_p$  processor-memory elements (PMEs) which communicate with one another via an interconnection network (ICN), as illustrated in Fig. 3.1. Each processor can reference data stored in its local memory ( $t_{loc}$  time units/reference) or, via the ICN, it can reference data stored in the memory of a remote PME ( $t_{rem}$  time units/reference,  $t_{rem} \geq t_{loc}$ ). While for some ICNs  $t_{rem}$  is a strong function of the relative positions of the PMEs on the network, for our purposes it is assumed that  $t_{rem}$  is a constant, upper bound on remote reference time. For networks with non-uniform routing distance, exploiting locality of reference is then an optimization that may be applied later.

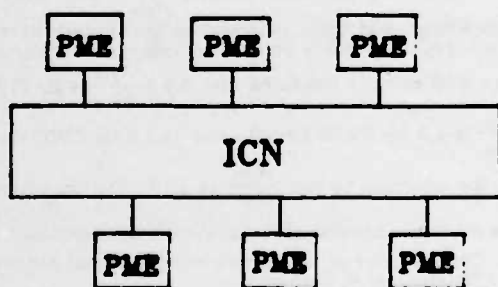


Fig.3.1 A Multiprocessor Configuration

#### 3.2 Partitioning the Problem

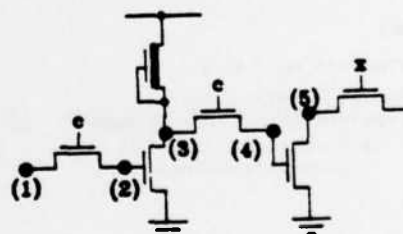
There are a number of ways that an algorithm such as ITA can be partitioned for implementation on a multiprocessor. The two basic partitioning techniques are:

- (1) **Functional Partitioning:** A conventional process-based approach; allocates distinct functions as processes to each processor either dynamically or statically. These functions can be allocated at the instruction level for a fine-grained approach, such as data-flow [26], or the functions can be coarse-grained, such as allocation of event scheduling to one processor, MOSFET processing to another, current summing to a third, and so on.

- (2) **Data Partitioning:** In this case, each processor performs a similar set of functions but on different data items which are allocated either dynamically or statically to each processor. In the electrical simulation case, this might correspond to allocating the evaluation of different collections of transistors to each processor while the steps performed to evaluate the transistors are common to all processors.

Unfortunately, coarse-grain functional partitioning does not lend itself to uniform growth - adding more PMEs may not lead to improved performance unless the functional units are connected by a flexible interconnection network, and the system is designed so that multiple copies of critical functions can be utilized effectively. This problem can often be solved by re-architecting the allocation of functions to processors but that is an expensive task and to be avoided if possible.

The approach described in this paper is based on data partitioning. Here a single sub-circuit described using an MNA matrix is allocated to each processor. To simplify the description of the algorithm, it is assumed that each sub-circuit consists of a single electrical node. It is also assumed for now that  $t_{rem}$  does not depend on network loading conditions. Consider the circuit fragment shown in Fig. 3.2(a) and the multiprocessor shown in Fig. 3.2(b). The nodes and their associated fanin elements have been allocated to specific PMEs. For this example, it is assumed the allocation is static however nodes could migrate to free PMEs dynamically, as described later. Note that there are more circuit nodes than PMEs, which is usually the case and is required to obtain maximum efficiency. Therefore, each PME is responsible for processing more than one electrical node; a separate event scheduler can be implemented on each PME for handling this situation or a process-level solution may be achieved by implementing virtual PMEs[27].



There are two principle ways in which the processor activity can be coordinated. We have categorized them as *explicit methods*, where a central scheduler is implemented on a single processor and coordinates the equation solution on the other processors, and *implicit methods*, where the scheduling is distributed and performed asynchronously. The method we have implemented at this time is an implicit approach. A single global variable, called *GlobalRemainingNets*, is used to coordinate the processors at a given time point. It is incremented whenever a node is scheduled at this time point and is decremented when a node has finished being processed. When *GlobalRemainingNets* reaches zero, all processors move to the next time point of the simulation. From the point of view of a single PME,  $P_i$ , once it has been allocated a set of electrical nodes,  $M_i$ , it proceeds as follows at time  $t_n$ :

```

foreach (node  $i$  in  $M$  scheduled at  $t_n$ ) {
  STEP (1): /*
    foreach (fanin element at  $i$ )
      obtain its fanin node voltages,  $v_j^f$ ,  $j \neq i$ ,  $K = k$  or  $k+1$ ;
  STEP (2): /*
    foreach (fanin element at  $i$ )
      compute its contributions to nodal equation;
    obtain  $v_i^{k+1}$  using a single Newton-Raphson step
    as described in Section 2;
    if (convergence is achieved) {
      if ( $v_{i,n} \neq v_{i,n-1}$ ) {
        schedule  $i$  at  $t_{n+1}$ ;
        decrement GlobalRemainingNets;
      }
    }
    else {
      schedule  $i$  again at  $t_n$ ;
      forall (fanout nodes of  $i$ ) {
        increment GlobalRemainingNets;
        send message to their PME to
        schedule fanout node at  $t_n$ ;
      }
    }
  }
}

```

*Fanin elements* of  $i$  are circuit elements (transistors, capacitors, voltage sources, logic gates, etc.) which are used to determine the new voltage at  $i$ , as illustrated in Fig. 3.3. On average, there are  $N_{FNI}$  fanin elements per node. To process each fanin element, it is necessary to obtain its controlling node, or *fanin node*, voltages,  $v_j^f$ . Assume there are, on average,  $N_{FNV}$  fanin node voltages that must be obtained per node iteration. *Fanout nodes* of  $i$  are defined as nodes with at least one fanin element connected to node  $i$ . There are an average of  $N_{FNO}$  fanout nodes per node. Of course, voltage supplies, clocks, and the ground node are not considered fanout nodes since they do not represent independent node voltages.

Using our simple model, the values of the controlling node voltages  $v_j^f$  will require a local memory reference per node if the node resides on the same PME as node  $i$ , otherwise it will require remote memory references. In the worst case, all fanin element nodes will reside on remote PMEs and all memory references will require  $t_{rem}$  units of time. Assuming only one remote memory reference can be active at any time for a particular PME, the average time taken for

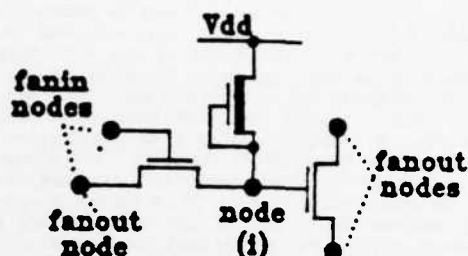


Fig.3.3 Circuit fragment showing fanin and fanout nodes

Step (1),  $t_1$ , can be approximated by:

$$t_1 = N_{FNI} t_{rem} \quad (3.1)$$

Step (2) does not require the processor to wait for a remote answer and hence the time taken in Step (2) depends only on the performance of the PME, not the ICN. The time required to solve a single nodal equation is proportional to the number of fanin elements, since each one must be processed for the Newton-Raphson step. In fact, the processing of each transistor,  $t_{trans}$ , dominates the PME time, with a small amount of time,  $t_{mod}$ , for checking convergence, updating local memory, etc. The time required for Step (2) can be written:

$$t_2 = t_{mod} + N_{FNI} t_{trans} \quad (3.2)$$

For MOS or Bipolar circuits, where each transistor has three controlling terminal voltages,  $\frac{N_{FNI}}{N_{FNV}} = 2$ . If supply voltages and ground are considered as special-case nodes and do not require remote reference, analysis of a number of large MOS circuits indicates that  $0.5 \leq \frac{N_{FNI}}{N_{FNV}} \leq 2$ . Typically  $\frac{N_{FNI}}{N_{FNV}}$  is 1.2 for NMOS circuits and is 1.5 for CMOS circuits. For the example circuit  $\frac{N_{FNI}}{N_{FNV}}$  is 1.16. The importance of

this result will become clear in later in this section.

One measure of the performance of a multiprocessor is its *efficiency*,  $\eta(N_p)$ , where:

$$\eta(N_p) = \frac{t_{sim}(1)}{N_p t_{sim}(N_p)} \quad (3.3)$$

where  $t_{sim}(N_p)$  is the "wall clock" simulation time using  $N_p$  PMEs.

Even on an ideal multiprocessor it is not possible to achieve an efficiency of 1.0 unless there are sufficient active electrical nodes available to keep all the PMEs busy at all times.

### 3.3 Ideal Models

Consider an *Ideal Gauss-Seidel Multiprocessor* where  $t_1 = 0$  (the ICN is infinitely fast) and  $t_2$  is constant for all electrical nodes. Such a machine also schedules nodes to PMEs using an optimal dynamic scheduling technique in zero time. Simulation of such a machine for our example circuit provides the following upper bounds for efficiency:



$N_p$	speedup	$\eta(N_p)$
1	1.00	1.00
2	1.97	0.98
3	2.92	0.97
4	3.81	0.95
5	4.80	0.96
6	5.65	0.94
7	6.51	0.93
8	7.32	0.92
9	8.06	0.90
10	8.86	0.89
11	9.56	0.87
12	10.26	0.85
13	10.93	0.84
14	11.65	0.83
15	12.26	0.82
16	12.95	0.81
32	20.46	0.64
64	26.47	0.44
128	34.16	0.27

Note that beyond 32 processors less than half the maximum performance of the machine can actually be achieved for this circuit.

A more realistic model for multiprocessor-based simulation would assume a constant delay in the ICN such that  $t_{comm} = t_{fixed}$ . With a random, static assignment of nodes to PMEs, the efficiency of such a multiprocessor for the example circuit is:

$N_p$	speedup	$\eta(N_p)$
1	1.00	1.00
2	1.94	0.97
3	2.81	0.94
4	3.71	0.93
5	4.53	0.91
6	5.44	0.91
7	6.15	0.88
8	6.78	0.85
9	7.45	0.83
10	8.14	0.81
11	8.82	0.80
12	9.35	0.78
13	9.84	0.76
14	10.36	0.74
15	10.86	0.73
16	11.64	0.73

The new model results shows a reduction in efficiency. However, the overall efficiency is still very high.

To obtain maximum efficiency, it is necessary to match the size and activity of the circuit under analysis to the number of processors it uses. Empirical results provide the best guidelines for such allocation on a real multiprocessor. Note that since the analysis is decoupled via the ITA algorithm, many independent circuits can be simulated at the same time on the same multiprocessor. By running a number of circuits, on a large number of processors, the overall efficiency can be kept high.

### 3.4 Allocation of Nodes to PMEs

Nodes can be allocated to PMEs either statically, before the analysis, or they can be allocated dynamically as the simulation proceeds. In either case, it is important that the time taken for determining the allocation is small otherwise it may dominate the total simulation time.

In the case of static allocation, a number of strategies are possible. These include:

- (1) **Random Allocation**, where the nodes are allocated to processors in random order while maintaining approximately the same number of nodes on each processor.
- (2) **Trace Allocation**, where nodes that are connected in a serial path (i.e. a fanout element of one node is a fanin element of the next) are stored on the same processor to minimize the number of off-processor memory references. Whenever more than one fanout element is present, the other fanout nodes are allocated to different processors.
- (3) **Minimum Distance Allocation** where, for ICNs where  $t_{comm}$  varies depending on where the remote processor is located on the network, adjacent electrical nodes are placed on adjacent processors on the ICN if possible.

Of these strategies, (1) involves the minimum amount of setup time.

In the optimal dynamic allocation scheme, nodes are allocated dynamically to processors to keep the load on all processors balanced. Note that this requires additional remote memory references or movement of the circuit description data as the simulation proceeds.

For the example circuit, the difference in efficiency between a random, static allocation and optimal dynamic allocation was less than 10% in almost all cases. For this reason, the overhead associated with more sophisticated allocation strategies may render them less efficient overall. The results reported in Section 4 are obtained from a random allocation strategy.

### 3.5 Circuit Partitioning

As mentioned earlier, circuits can be partitioned into sub-circuits, where the elements of each sub-circuit are strongly connected. This partitioning serves two purposes. First, it permits direct methods to be used for the tightly-coupled portions of the circuit, resulting in a more efficient analysis. Since the sub-circuits are highly connected, they can be processed using full matrix techniques which are amenable to speed-up via pipelining.

Second, by allocating an entire sub-circuit to each processor the amount of computation performed locally can be adjusted to balance PME and ICN loads.

### 3.6 Choice of ICN

Many different intercommunication networks have been developed [28,29] but only a few meet the requirements of this application. In particular, ICNs based on the Perfect Shuffle connection [30,31] have the following desirable properties:

- (1) **Low latency**, which grows as the log of the number of ports in the ICN.
- (2) **Constant switch element size**. The number of ports on each switch element is independent of the size of the ICN.
- (3) **Uniform Loading**. All network elements see a similar load.
- (4) **Easy Growth**. Only wiring changes and identical switches are necessary to increase the number of PMEs on the multiprocessor.
- (5) **Simple routing algorithm**. All routing decisions can be made on the basis of local information and the routing process is very fast.

Because of the high performance of a Shuffle-based ICN, high electrical simulation efficiencies can be achieved. For a large circuit containing  $N$  electrical nodes with  $C$  nodes actively changing at any time,  $C \leq N$ , the total time spent solving the independent node equations on a serial computer is approximately:

$$T_s(C) = C(t_{\text{total}} + \bar{N}_{\text{PME}} t_{\text{eval}}) \quad (3.4)$$

Consider a multiprocessor using a single or multiple stage Shuffle network with an element cycle time of  $t_c$  and a latency  $k \log(N_p)$ , where  $N_p$  is the number of PMEs connected via the shuffle network and  $k$  is a constant. Then  $t_{\text{eval}} = t_c k \log(N_p)$ . For now assume  $N_p > C$ , then the total analysis time on such a network is approximately:

$$T_p(C) = (t_{\text{total}} + \bar{N}_{\text{PME}} t_{\text{eval}}) + \bar{N}_{\text{PIN}} t_c k \log(N_p) \quad (3.5)$$

Eqn. (3.5) is in fact a worst-case figure because it assumes all communication is on the critical path of the computation and that there is no pipelining of requests. The speed-up factor for the parallel computation is then:

$$\frac{T_s}{T_p} = \frac{C(t_{\text{total}} + \bar{N}_{\text{PME}} t_{\text{eval}})}{(t_{\text{total}} + \bar{N}_{\text{PME}} t_{\text{eval}}) + \bar{N}_{\text{PIN}} t_c k \log(N_p)} \quad (4.3)$$

If  $k$  is from 1 to 3[31]; if  $C \approx N_p$ ; with  $\frac{\bar{N}_{\text{PIN}}}{\bar{N}_{\text{PME}}} \approx 1.2-1.5$  as shown above and  $t_{\text{total}}$  negligible, then if  $t_{\text{eval}} \approx t_{\text{total}}$ , the speed-up becomes approximately  $\frac{C}{1 + \log C}$ . However, if the equation solution time is larger than the network cycle time, then the speed-up factor will be closer to  $C$ , as is demonstrated in Section 4.

### 3.7 The MSPLICE program

The MSPLICE program was developed to verify the above model. It uses the Gauss-Seidel-Newton algorithm described earlier, but can also be run using a less constrained variation of the algorithm known as *weakly chaotic relaxation*. In this case, a PME can continue to solve a node for iterations  $k+1, k+2, \dots, k+F_m$ , where  $F_m$  is the maximum number of iterations a node can move ahead before it must wait for updates of the values of its fanin node voltages. For Gauss-Seidel-Newton,  $F_m = 1$ .

The algorithm used in the MSPLICE program proceeds as follows:

```

ITAControlLoop()
while( GlobalMoreToGoFlag is TRUE )
    WaitMessage( DoneAtQueue, processorNumber, flag );

    if( flag is TRUE )
        increment remainingProcessors;
    else
        decrement remainingProcessors;

    if( (remainingProcessors is 0) and (GlobalRemainingNets is 0) )
        if( GlobalFutureActivityFlag is FALSE )
            GlobalMoreToGoFlag = FALSE;
        else
            /*
             * All processors must be blocked after having finished
             * all the work they had at this time.
             * Tell them that there is no more work to go on to the next time period.
             */
            GlobalFutureActivityFlag = FALSE;

            forall( processor in processors )
                q = processorControlQueue( processor );
                SendMessage( q, TRUE );

ITAMainProcessorLoop()
while( GlobalMoreToGo is TRUE )
    forall( evaluationRequest in elementEvaluationRequestQueue )
        ITAProcessElementEvaluationRequest( evaluationRequest );

    forall( evaluationReply in elementEvaluationReplyQueue )
        ITAProcessElementEvaluationReply( evaluationReply );

```

```

forall( net in netEvaluationRequestQueue )
    ITAProcessNetEvaluationRequest( net );

if( all queues are empty )
    /*
     * wait for the global counter to go to zero or to get more input
     * requests at this time.
     */

    SendMessage( DoneAtQueue, MyProcessorNumber, TRUE );

    wait( all queues )
    if( more input requests at this time )
        SendMessage( DoneAtQueue, MyProcessorNumber, FALSE );
    else
        T = T + 1;
        swap evaluation queues for T and T+1;
        schedule any input sources at T+1;

ITAProcessElementEvaluationRequest( evaluationRequest )
{
    net = evaluationRequest->net;
    element = evaluationRequest->element;
    norton = ITACompanionNode( net, element );
    q = elementEvaluationReplyQueue( home_processor( net ) );
    reply->net = net;
    reply->norton = norton;
}

ITAProcessElementEvaluationReply( evaluationReply )
{
    net = evaluationReply->net;
    faninNorton = evaluationReply->norton;
    net->norton = net->norton + faninNorton;
    net->remainingFanins = net->remainingFanins - 1;

    if( net->remainingFanins is 0 )
        ITACheckConvergence( net );
}

ITAProcessNetEvaluationRequest( net )
{
    forall( faninElement in net )
    {
        q = elementEvaluationQueue( home_processor( faninElement ) );
        request->net = net;
        request->element = faninElement;
        SendMessage( q, request );
    }

    /*
     * Check convergence, schedule self and fanouts, and keep GlobalRemainingNets
     * (the global count of the number of nets under evaluation) consistent.
     */

    ITACheckConvergence( net )
    {
        if( net has converged at this time )
        {
            if( net has not previously converged at this time point )
            {
                if( change from the previous time point is significant )
                {
                    schedule the net at T+1;
                    if( T+1 queue has just become non-empty )
                        GlobalFutureActivityFlag = TRUE;
                }
                schedule the net's fanouts at this time point;
                increment GlobalRemainingNets by Nfanouts-1;
            }
            else
                re-schedule the net at this time;
            decrement GlobalRemainingNets;
        }
    }
}

```

Here,  $T$  represents the current value of simulated time in units of Minimum Resolvable Time (MRT)[21], and the forall construct means execute the block on all members of the set in any order and, therefore, they may be executed concurrently.

MSPLICE has been implemented on both the Digital VAX11-780 and the the BBN Butterfly machine.

## 4. EXPERIMENTAL IMPLEMENTATION

**4.1 The Hardware** In order to evaluate the performance of MSPLICE, it was necessary to use a real multiprocessor. The IBM Butterfly[32,33] multiprocessor was chosen for these experiments as it was the only machine available whose characteristics approximate those described above.

The IBM Butterfly is a tightly-coupled multiprocessor. An entire system may contain from 1 to 256 processor nodes. Each processor node is itself a multiprocessor, consisting of a Motorola 68000 and an AMD 2901-bit-slice-based Processor Node Controller (PNC). Together, these processors can provide from 1/2-1/3 the integer performance of a DEC VAX-11/780 on certain examples. Once the high-performance floating-point co-processor board that we are developing for the machine has been completed, floating-point performance in the same range is expected. At this time, however, floating-point is implemented at the assembler level and is relatively slow.

All memory in the system is associated with the processor boards. Each processor board contains 256K bytes of memory, with an optional memory expansion interface capable of holding 4M bytes.

The computation and I/O processors are connected by a network consisting of  $\log_4 n$  stages of shuffle-exchange. This network is also known as a Base 4 Omega Network. When the 68000 attempts to read or write a virtual address which represents memory on a remote processor board, the local 2901 communications co-processor, and the 2901 communications co-processor on the remote board work together to read or write the correct location. Together, the PNC's and the switch provide transparent remote memory access across the entire machine with uniform low latency and high bandwidth. Local memory reference are completed in 625ns. Single word read or writes from any processor to any remote memory are normally completed in under 4 $\mu$ s on a 16 processor configuration, and under 5 $\mu$ s in a 256 processor configuration. This results in  $\frac{t_{\text{remote}}}{t_{\text{local}}} \approx 8$ , which is comparable to the ratio of a cache miss to a cache hit on a uniprocessor. The low cost of remote memory references makes the Butterfly attractive for our application.

Block transfers are performed at a rate of 32 Mbits/second. Because the structure of the Butterfly switch does not correspond to that of a fully-connected graph, it cannot provide every possible pattern of connections between inputs and outputs in a single operation. However, because of the high performance of the switch and the statistical frequency of conflicting requests, this does not appear to be a problem, and a simple back off-retry scheme is used whenever an access conflict occurs inside the switch.

### 4.2 The Programming Environment

The operating system for the Butterfly, called Chrysalis[34], is a simple operating system written in the 'C' programming language[35]. Access to shared software resources in the machine is provided by object handles which are global identifiers and are unique throughout the machine. Whenever a process needs access to such a resource, it uses Chrysalis to map the object into its virtual address space.

Chrysalis is oriented around the concept of process-level concurrency and provides segmentation-based virtual memory management, and the ability to create, destroy processes. Process images are loaded dynamically on demand to minimize the amount of communication with the host system.

Dual queues are also provided to allow efficient locking of resources and passing of data between processes without explicit locking.

## 4.3 Experimental Results

The results reported in this section were obtained on a ten-processor machine. For the test circuit, the following speed-up and efficiency values were obtained using MSPLICE:

$N_p$	speedup	$\eta(N_p)$
1	1.00	1.00
2	1.83	0.92
3	2.28	0.76
4	3.40	0.85
5	4.09	0.82
6	4.55	0.76
7	5.33	0.76
8	5.98	0.75
9	6.77	0.75
10	7.04	0.70

Note that for nine processors the speedup and efficiency of MSPLICE on the Butterfly was 90% of the maximum possible for this example, as presented in Section 3.

The overall run time of MSPLICE is reduced due to the poor floating point performance of the Butterfly. In fact, with  $t_{\text{float}} \sim 1.6\text{ms}$  on the Butterfly for the simple MOS model used in MSPLICE,  $\frac{t_{\text{float}}}{t_{\text{local}}} \approx 400$ . However,  $t_{\text{float}}$  for the same model on a VAX-11/780 with FPA is approximately 300 $\mu$ s. On the other hand,  $t_{\text{float}}$  for a modern MOS model, which considers short channel and other complex effects, on the VAX is approximately 8ms. Therefore, for such an MOS model, even with 50MIP processors on the Butterfly nodes  $t_{\text{float}} > 10 t_{\text{local}}$  and high efficiency will be maintained.

A conservative estimate, with 10MIP PNEs,  $N_p = 256$ ,  $\eta(256) = 0.5$  for a 25,000 node circuit ( $\sim 70,000$  MOSFETS), results in 1.3GIP performance. In other words, the analysis of a 70,000 MOSFET circuit on this processor would take about the same time as the same analysis of a 50 MOSFET circuit on a VAX-11/780.

## 5. SUMMARY

The implementation of a new form of relaxation-based electrical circuit simulation for a special-purpose multiprocessor has been presented. This technique promises to improve the speed of accurate, electrical circuit simulation dramatically while reducing the overall cost of the analysis.

An implicit algorithm for the implementation of the ITA simulation method has been described and experimental results from its use on the Butterfly multiprocessor have been presented. This data confirms our analysis that the use of a high-performance, Perfect Shuffle based intercommunication network, in conjunction with the distributed ITA algorithm, results in a highly efficient utilization of the multiprocessor hardware.

While the use of these techniques for circuit simulation has been presented, the approach described here may be applied to other electrical network graph-based problems with similar speed improvement characteristics.

## ACKNOWLEDGMENTS

We thank R. Rettberg, F. Hart, and J. Goodhue of Bolt, Beranek, and Newman for their strong support of this project. We also thank R. Saleh, A. Sangiovanni-Vincentelli, and J. White for many helpful discussions and D. Cheng for the design of the first Butterfly floating-point co-processor. This work was supported in part by the Army Research Office under contract DAAG29-81-K-0021 and by the Semiconductor Research Corporation. Recent support from DARPA under grant N00039-83-C-0107 is also gratefully acknowledged.



## The TimberWolf Placement and Routing Package

Carl Sechen and Alberto Sangiovanni-Vincentelli

Department of EECS  
University of California  
Berkeley, California 94720

### Abstract

TimberWolf is an integrated set of placement and routing optimization programs. The general combinatorial optimization technique known as simulated annealing [1] is used by each program. Programs for gate array, standard cell, and macro/custom cell placement, as well as standard cell global routing have been developed. Experimental results on industrial circuits show that area savings over existing layout programs ranging from 15 to 40% are possible.

### 1. Introduction

TimberWolf is an integrated set of placement and routing optimization programs. The general combinatorial optimization technique known as simulated annealing [1] is used by each program. Four basic optimization programs of the TimberWolf package have been developed.

(1) A standard-cell placement program. This program places standard cells into rows and/or columns in addition to allowing user-specified macro blocks and pads. The program was interfaced to the CIPAR standard cell placement package developed by American Microsystems, Inc. For larger circuits (800 to 1500 cells), TimberWolf reduced total wire length from 45 to 57% in comparison with CIPAR alone. Furthermore, final chip areas were reduced by at least 30%, as a result of the improved placement. For a circuit of 1000 cells, TimberWolf reduced the final chip area by 31% in comparison to CIPAR and by 21% over another commercially available standard cell placement program.

(2) A standard cell global router program. The global router reduced by 10 to 15% the number of wiring tracks used by the CIPAR router. This translated to an overall area savings of 5 to 7%. Vecchi and Kirkpatrick [2] recently described the use of simulated annealing for global routing.

(3) A generalized gate-array placement program allowing user-specified macros and primary terminals. This program found placements with a 6 to 27% reduction in total estimated wire length for several benchmark problems in comparison to the best published results. This program optionally includes in the cost calculation a measure of the local routing congestion.

(4) A macro/custom cell placement program. This program places cells of any rectilinear shape. Furthermore, the cells may have fixed geometry including pin locations (macro cells) or they may have fixed area with a given aspect ratio range and with pins that need to be placed (custom cells). All rotations and reflections of each cell are considered. TimberWolf also has the ability to place cells among user-defined sub-regions of the chip. TimberWolf allows multiple chips to be placed simultaneously. This package can also be used to place circuits on one or more printed circuit boards.

### 2. The Basic Algorithm

Simulated annealing [1] is a statistical method for solving general combinatorial optimization problems formulated as follows. Given a problem specified by a state  $x$  which is an element of a discrete set  $X$  and a cost function  $c(x)$ , find  $x^*$  such that  $c(x^*)$  is minimal.

#### 2.1 Algorithm Structure

The following function gives the general structure of the algorithm.

```
structuredAlgo(x, c) {  
    /* state x and cost function. c */  
    while( "stopping criterion" is not satisfied ) {  
        generate a new state x';  
        evaluate c(x'); /* new cost */  
        if( accept( c(x'), c(x) ) )  
            /*  
                accept returns 1 if an  
                acceptance criterion has been  
                satisfied and 0 otherwise  
            */  
            x = x';  
    }  
}
```

Note that the important part of the algorithm is the function `accept`. Simulated annealing uses a statistical method to decide whether to accept or reject the new state. A parameter  $T$  and a random number generator are critical to the strategy.

```
accept( c(x'), c(x) ) {  
    /*  
        given the cost of a new state x' and of  
        the previous state, x, return 1 if the cost  
        variation passes a test. T is a parameter  
    */  
     $\Delta c = c(x') - c(x)$ ;  
    if(  $\Delta c \leq 0$  ) {  
        return( 1 );  
    } else {  
         $y = \exp(-\Delta c / T)$ ;  
         $r = \text{random}(0, 1)$ ;  
        /*  
            random is a function which returns a pseudo  
            random number between 0 and 1 (with  
            uniform distribution).  
        */  
        if(  $r < y$  ) {  
            return( 1 );  
        } else {  
            return( 0 );  
        }  
    }  
}
```

In the function `accept`, note that new states characterized by  $\Delta c \leq 0$  always satisfy the acceptance criterion. However, for the new states characterized by  $\Delta c > 0$ , the parameter  $T$  plays a fundamental role. If  $T$  is very large, then  $r$  is likely to be less than  $y$  and a new state is almost always accepted irrespective of  $\Delta c$ . If  $T$  is small, close to 0, then only new states that are characterized by very small  $\Delta c > 0$  have any chance of being accepted. In general, all states with  $\Delta c > 0$  have smaller chances of satisfying the test for smaller values of  $T$ .

Simulated annealing as proposed by Kirkpatrick, Gelatt, and Vecchi [1] incorporates an automatic mechanism to adjust the value of  $T$  during the optimization process. In addition, new states are generated by a random process. The general form of the simulated annealing algorithm follows.

```

simulatedAnneal(x, T) {
  /* Given an initial state x and initial parameter T. */
  while( "stopping criterion" is not satisfied ) {
    generate T < T;
    T = T;
    while( "inner loop criterion" is not satisfied ) {
      generate a new state x';
      evaluate the new cost c(x');
      if( accept( c(x), c(x') ) )
        x = x';
    }
  }
}

```

Simulated annealing was developed as an analogy to physical annealing. The best results with simulated annealing are obtained by starting with a large value of the parameter  $T$ , whereby virtually all proposed new states are accepted. Further, the best results are obtained when the system is allowed to achieve equilibrium at each stage of the annealing process (that is, for each value of  $T$ ). This is implemented by the "inner loop criterion" in the simulated annealing algorithm. The "stopping criterion" is satisfied when the cost function's value remains the same after several stages of the annealing process.

In simulated annealing, the best results are obtained when the parameter  $T$  is slowly reduced when the cost function's value begins to decrease significantly. For each successive step of the annealing process,  $T$  is lowered exponentially. That is,  $T = T \cdot \alpha$ , with  $0 < \alpha < 1$ . The parameter  $\alpha$  can be a constant or can also be a function of  $T$ . The TimberWolf programs currently allow the value of  $\alpha$  to be specified for each value of  $T$ . The value of  $\alpha$  is usually in the range of 0.8 to 0.95.

## 2.2 The TimberWolf Implementation of the Simulated Annealing Algorithm

### 2.2.1 Generating New States

The TimberWolf programs begin with a random initial placement or wiring configuration. A new state is generated by either exchanging two fundamental units or moving a unit to another location. For the gate array placement program, the new state is generated by the interchange of two modules, where a module refers to a fundamental unit specified in the net list. The standard cell placement program also generates new states by the interchange of cells. However, because standard cells typically vary in width, the interchange of two cells often results in a non-feasible solution because overlaps are not allowed. This is solved by a penalty function approach, first described by Kirkpatrick, Gelatt, and Vecchi [1]. The TimberWolf implementation of this approach will be described in the next section. The penalty function approach is also employed by the macro/custom cell placement program because the cells typically vary in both height and width.

For the standard cell and macro/custom cell problems, new states are also generated by the movement of a cell to a new location. Experimental investigation has revealed that the use of both methods of generating new states is necessary to achieve the best results. Furthermore, orientation changes of standard and macro/custom cells are performed which result in new states. New states are also generated for custom cells by assigned a new location to a pin or group of pins.

For the standard cell global router program, new states are generated by assigning a portion of a net to a different channel.

### 2.2.2 Cost Function

The cost function for the placement programs is based on total estimated wire length. The standard cell and macro/custom cell programs also include a penalty function term. The cost function for the standard cell global router is based on the estimated wiring area which is approximated by the sum over all channels of the channel density.

### 2.2.3 Generating New Values of T

In the current implementation of TimberWolf, the parameter  $\alpha$  is user-specified as  $\alpha$  versus  $T$  data. The best results have been obtained when  $\alpha$  is the largest (approximately 0.95) during the stages of the algorithm when the cost function is decreasing rapidly. Furthermore, the value of  $\alpha$  is given its lowest values at the initial and latter stages of the algorithm (usually 0.80). The value of  $\alpha$  is gradually increased from its lowest value to its highest value, and then gradually decreased back to its lowest value.

### 2.2.4 The Inner Loop Criterion

The inner loop criterion is implemented by the specification of the number of new states generated for each stage of the annealing process. This number is specified as a multiple of the number of fundamental units for the placement or routing problem. For the gate array placement and standard cell global router programs, 20 new states per unit are generated at each stage. The standard cell and macro/custom cell placement problems have many more degrees of freedom (orientation changes, pin location changes, etc.) and hence 100 or more new states are generated per cell at each stage.

### 2.2.5 The Stopping Criterion

The stopping criterion is implemented by recording the cost function's value at the end of each stage of the annealing process. The stopping criterion is satisfied when the cost function's value has not changed for 4 consecutive stages.

## 3. Standard Cell Placement Optimization Program

### 3.1 Introduction

This program optimizes the placement of standard cells into row and/or column blocks. Furthermore, the various blocks may have differing heights. The program also optimizes the placement of pads or buffer circuitry, as well as macro blocks. The macro blocks may be positioned anywhere on the chip. The estimation of the wire length for a single net is determined by computing the half-perimeter of the bounding box of the net. The bounding box is defined by the smallest rectangle which encloses all of the pins comprising the net. For the case of a two-pin net, this is the Manhattan distance. Because exact pin locations are used in the wire length calculations, TimberWolf considers all possible orientations for a cell, pad, or macro block. Pins which are internally connected within a cell are treated as a single pin with a location which is the average of the locations of its constituent pins.

The program employs the exchange class mechanism for blocks as well as cells, pads and macros. If two blocks have the same exchange class, then cells from these blocks are interchangeable. Blocks with differing exchange classes may not have their cells interchanged. Differing exchange classes for blocks are usually employed when blocks have different heights. Furthermore, two cells or two pads may be interchanged only if they belong to the same exchange class.

### 3.2 Algorithm Details

The cost function for the simulated annealing algorithm consists of two independent portions. The first portion is the total

estimated wire length. The second portion is the penalty function which consists of a total sum of overlap penalties. This penalty function was incorporated because of the usual difference in width of the standard cells. Often two cells are selected for interchange which differ in width. Therefore, an exchange of location of these two cells often results in some overlap with one or more of the other cells. Furthermore, the program often selects a single cell for a displacement to a new location. Once again, some overlap may result. The exchange of cells or the displacement of a single cell may also result in a portion of a cell dangling off the end of a row or column block. This is treated as a case of overlap with an imaginary cell being located at the ends of each column and row block. This feature increases the number of states in the state space  $X$ . Experimental investigation has shown that this results in better placements.

When two standard cells overlap, a penalty is assessed which is proportional to the square of quantity of the amount of overlap plus an offset parameter. The offset parameter is chosen to ensure that when the parameter  $T$  approaches zero, then the total amount of overlap approaches zero.

The alternative to the aforementioned overlap concept is of course to not allow overlaps. For example, when inserting a cell into a row block, if insufficient space is available then the cells to the right are all shifted farther to the right as necessary. This has the obvious disadvantage of destroying the relationships between the shifted cells and the cells on the neighboring rows. The overlap concept was employed so as to not disturb the placement of the remaining cells when performing an interchange of cells or a displacement of a single cell.

The selection of new states is based on the following considerations: (1) A random number between one and the total number of cells, pads and macro blocks is generated. The cells are numbered from one to the number of cells, and the pads and macro blocks are numbered starting from the number of cells plus one. If the random number is less than or equal to the number of cells, then a cell is selected. Otherwise, a pad or macro block is selected. (2) A second random number is selected between 1 and the total number of cells, pads, and macro blocks. (3) If the two numbers selected both represent cells, then the pair of cells are interchanged to generate a new state. (4) Similarly, if two pads or two macro blocks were selected, then an interchange constitutes the new state. (5) If the two numbers selected do not represent the same unit (that is, cell, pad, or macro block) then the first unit selected governs the generation of a new state. If this first unit was a cell, then this cell is displaced to a new location. If this new state is rejected, then the next state generated is an orientation change for the cell. If the first unit was a pad or macro block, then an orientation change of the respective unit is attempted.

The ratio of single cell displacements to cell interchanges has a pronounced effect on the quality of the final placement. Experimental investigation has revealed that a ratio of about 5 to 1 yields the best results. Hence, if the first unit selected was a cell, the generation of the second random number is weighted to produce the desired ratio. This is implemented by generating a random number between one and the number of cells multiplied by 5.

In the latter stages of the algorithm, that is, when the value of  $T$  approaches zero, the displacement of a cell has very little chance of being accepted unless the displacement is very local. Similarly, an exchange of distant cells has a vanishingly small chance of being accepted. Hence, it is more efficient to employ a range limiter, which limits the range of the displacement of a cell or cells. Consequently, during the latter stages of the algorithm, the cells undergo many small displacements while gradually eliminating overlaps and reducing wire length.

Of major concern to all implementations of the simulated annealing algorithm is CPU time. The TimberWolf standard cell program was designed to reduce computation time while sacrificing storage. One of the features of the program is that computation time per iteration is constant (that is, it is invariant with the number of cells). The iteration time is defined to be the time required to generate a new configuration, evaluate the new value of the cost function, and then decide to accept or reject the new configuration. Two key features make this possible: (1) The cells in a block are hashed into bins that partition the block's coordinate system. Hence overlap calculations require a constant amount of time. (2) The possible orientations for a cell, including the pin locations for each orientation, are computed at the outset and are

stored. Thus, to change a cell orientation, only a pointer change is required rather than recomputing the cell boundaries and pin locations.

Many current standard cell optimization programs attempt to first perform an inter-row optimization and then an intra-row optimization. That is, each cell is first assigned to a row and then in a second step, the cells are placed within their respective row. The method employed by TimberWolf simultaneously considers both optimizations, thus yielding much reduced routing area.

### 3.3 Results

The program was interfaced to the CIPAR standard cell placement package developed by American Microsystems, Inc. For the larger circuits (800 to 1500 cells), TimberWolf reduced total wire length from 45 to 57% in comparison with CIPAR. Furthermore, final chip areas were reduced by at least 30%. For a circuit of 1000 cells, TimberWolf reduced the final chip area by 31% in comparison to CIPAR and by 21% over another standard cell place and route package marketed by a workstation company.

The computation time was 4 milliseconds per iteration (VAX 11/780 running VMS). The memory requirement is linearly related to the number of cells. For the largest circuit (1500 cells), 20 million iterations were performed for some of the better placements. This implies nearly 24 hours of CPU time. The memory requirement for this circuit was 2 megabytes (32-bit integers are used). The results are summarized in Table 1.

The ITT and QUALIC circuits could not have their areas reduced more than 15% due to pad limitation. There are two versions of the TELEBIT circuit. The second version has very many of

Circuit	# Cells	Total Wire Length Reduction	Final Chip Area Reduction	CPU Time in Hours
TELEBIT1	1500	45%	30%	24
TELEBIT2	1500	37%	25%	12
XEROX	1000	57%	31%	8
ITT	200	41%	15%*	2
QUALIC	100	37%	15%*	0.5

\* denotes pad-limited

Table 1. Summary of Results  
TimberWolf Standard Cell Placement Optimization Program  
Circuits Provided Through the Courtesy of  
American Microsystems, Inc.

its cells specified to occur in fixed sequences. Hence the number of states  $x$  in the state space  $X$  is significantly reduced. It has been experimentally observed that a reduction of the cardinality of the state space results in a lower wiring area reduction.

The effect of the TimberWolf placement optimization can be further demonstrated by the number of route-through cells which were required. A route-through cell has two internally connected pins, one on the top and one on the bottom. If a portion of a net must connect two cells which are not on the same row and are not on neighboring rows, then this net must be routed through the rows between those containing the cells. A route-through cell must be inserted to accomplish this for the case of two levels of interconnect.

For the QUALIC circuit, the number of route-through cells was reduced from 50 to 14. Furthermore, the number of route-throughs was reduced from 51 to zero for the ITT circuit. For the XEROX circuit, more than 1000 route-through cells were eliminated. All of the approximately 300 route-through cells were eliminated for the 1500-cell TELEBIT circuit.

## 4. Standard Cell Global Router Program

### 4.1 Introduction

The layout of a standard cell circuit often consists of rows of cells bordered by pads and/or buffer circuitry. In order to minim-

ize the need for route-through cells (which increase the area of a circuit), the cells are typically designed with electrically equivalent (internally connected) pins on both the top and bottom side. Thus a net from above can be connected to the top pin while the same net from below can be connected to the bottom pin. The internally connected pins are referred to as a *pin cluster*. A portion of a net which must connect two pin clusters is referred to as a *net segment*.

It often arises that a pin cluster from one cell must be connected to a pin cluster from another cell on the same row. If each such cluster has a top pin and a bottom pin, then this net segment is defined as being *switchable*. A decision must be made as to whether to route the switchable net segment in the channel above or below the row. The TimberWolf global router assigns switchable net segments to channels based on the minimization of the *total channel density*. The total channel density is defined to be the sum of the channel densities for all of the channels.

The TimberWolf global router routes all nets and considers all pins except those nets and pins which route power and ground. It is often the case (as with CIPAR) that separate routines are used to route power and ground. The global router takes into consideration pins on the outer pads or buffer cells.

Some standard cell place and route systems (for example, CIPAR) do not employ a global router. Instead, only a channel router is used and it routes as many connections as possible for each channel. Thus the order in which the channels are routed can have a substantial effect on the total number of wiring tracks required (and thus the area of the circuit). In contrast, after using the TimberWolf global router, specific pins have been identified for interconnection. Thus the number of wiring tracks required is independent of the order in which the channels are routed.

## 4.2 Global Router Algorithm

The TimberWolf global router performs the optimization in two stages. The first stage examines each net separately. Two basic steps are applied to each net. (1) The first step identifies which pairs of pin clusters are to be connected based on the minimization of the Manhattan interconnection distance. This results in the identification of the net segments. (2) The second step considers each net segment and selects a pin from each cluster such that the Manhattan length of the segment is minimized. Two pairs of pins are selected for each switchable net segment.

The second stage results in the assignment of a channel for each switchable net segment. The two stages are detailed below.

### 4.2.1 First Stage of the Global Router Algorithm

The first stage consists of applying the two steps detailed below to each net separately.

#### Step 1

For a given net, the pin clusters that need to be connected are determined. A graph is formed in which the clusters are represented by the nodes and connections between the nodes (the formation of potential net segments) are represented by edges. An edge connects two nodes if a net segment could possibly connect the two clusters. For example, two clusters can be connected only if one of the following two conditions is true. (1) They lie on the same row, with no intervening cluster occupying the same row. This is the case of a potential switchable net segment. The net segment is switchable if each cluster has a pin on the top and on the bottom of the row. That is, the net segment could be routed either in the channel above the row or in the channel below the row. (2) They lie on neighboring rows. Furthermore, there cannot be another cluster lying between the two clusters which occupies either of the rows occupied by the two clusters.

The result of conditions (1) and (2) above is that the maximum degree of a node is 4. Further, this maximum degree is achieved when a given cluster is to be connected to two clusters in the row above (one to the left and one to the right) and to two clusters in the row below (also one to the left and one to the right).

The minimum spanning tree is generated for the graph via Kruskal's algorithm [3]. This portion of the algorithm effectively generates a Steiner tree [4] for the interconnection of the clusters. When the minimum spanning tree has been generated, pairs of pin clusters have been identified which are to be connected by a net segment.

#### Step 2

In this step, each edge of the minimum spanning tree is examined, and one pin from each cluster is selected to form the actual net segment. In the case of an edge connecting two clusters on the same row, it is determined if this is a switchable net segment. If the segment is switchable, then two pairs of pins are selected. One pair is for the segment routed in the channel above the row and another pair is for the segment routed in the channel below the row.

Pin selection proceeds as follows. (1) For the case of two clusters on neighboring rows, the bottom pin of the top cluster and the top pin of the bottom cluster are selected based on the minimization of the Manhattan distance between the two points. (2) For the case of two clusters on the same row: (a) If the edge is determined to be switchable, the top pin from each cluster is selected based on the minimization of the distance between the two points. Also, the bottom pin from each cluster is similarly selected. (b) If the edge is not switchable, either the pair of top pins (if the segment must be routed in the channel above the row) or the pair of bottom pins (if the segment must be routed in the channel below the row) are selected. The pin selection is again based on the minimization of the segment length.

### 4.2.2 Second Stage of the Global Router Algorithm

This step employs a simulated annealing algorithm. The net segments (for all of the nets) with their respective pins are supplied as input. One half of the minimum contact-to-contact spacing is added to each end of the horizontal span of each segment. For each switchable segment, an arbitrary initial selection (of above or below the row) is made. Each channel is examined sequentially to determine its density. The densities of the channels are summed, and this sum is the initial value of the cost function. A new state of the configuration is generated by the random selection of a switchable segment and then routing it on the opposite side of the row from its current position. As a result of the new state, the cost function either increases by 1, decreases by 1, or remains the same. That is, the total channel density changes by at most 1.

The case of no change in the cost is treated further. This is the case in which the net segment switch has no effect on the total channel density. A second cost function is introduced in this case. This cost function is a measure of the congestion in a channel between the two points defining the span of a net segment. The cost function is evaluated by taking the difference between the overall channel density and the density between the two points defining the span. The cost function is first evaluated for the span of the net segment in the original channel. Next, the cost function is evaluated for the net segment span in the new channel. The difference in cost ( $\Delta c$ ) is determined by subtracting the second cost function value from the first. A negative value of  $\Delta c$  indicates that switching the net segment to the new channel places the segment in a channel of less congestion.

### 4.3 Results

The global router reduced the number of wiring tracks used by the CIPAR router by 10 to 15%. Because routing typically occupies one half of the chip area, this translated to an overall area savings of 5 to 7%.

The global router was applied to the QUALIC circuit after a TimberWolf placement optimization. The total number of wiring tracks used without the global router was 72. After employing the global router, 65 tracks were used. This represents a 10% reduction in wiring area.

The largest circuit (1500-cell TELEBIT) was tested with and without TimberWolf placement. After the optimized placement, the



global router reduced the overall chip area by 6.1%. With only CIPAR placement, the area reduction was 5.5%. A total area savings of 34% was achieved for the TELEBIT circuit when both TimberWolf placement optimization and the global router were applied. The results are summarized in Table 2.

Circuit	# Cells	Global Router Area Reduction	Final Chip Area Reduction	Global Router CPU Time in Hours
TELEBITa	1500	6.1%	34%	2
TELEBITb	1500	5.5%	—	2
XEROX	1000	6%	35%	1
QUALIC	100	10%	—	0.2

Table 2. Summary of Results  
TimberWolf Standard Cell Optimization Programs  
Circuits Provided Through the Courtesy of  
American Microsystems, Inc.

The TELEBIT circuits that were tested did not have any cells belonging to fixed sequences. TELEBITa refers to a TimberWolf-placed circuit and TELEBITb refers to a CIPAR-placed circuit. The additional area reduction value for the QUALIC circuit represents the wiring area reduction, since the overall chip area was pad limited. The XEROX circuit was placed using TimberWolf.

## 5. Gate Array Placement Optimization Program

### 5.1 Introduction

This section describes the generalized gate array placement program. Each fundamental unit in a gate array will be referred to as a cell. Hence, a 50 by 50 gate array is said to have 2500 cells. Some gate array designs allow additional flexibility and hence greater gate utilization by creating functionally independent units within a cell. For example, Tektronix gate arrays widely utilize functional units which are half-cell sized. TimberWolf allows the functional units to be half-cell sized or quarter-cell sized. The term module will refer to a fundamental unit specified in the net list. A module may be the size of: (1) a full cell, (2) a half cell or (3) a quarter cell. Additionally, macro modules may be specified. A macro module consists of a pre-wired, arbitrarily-shaped collection of cells.

TimberWolf has other features which provide additional flexibility. For example, a module (or macro module) may be designated as unmoveable (that is, preplaced) or as belonging to an exchange class of modules. The modules in such a class may only be interchanged among themselves. This feature is often desirable when a group of modules on the edge of the gate array are to be considered as primary terminals. Often, the exact location of a given primary terminal is not important, only that it be on a given edge.

It is often the case that gate arrays have wider channels in the center of the array. This is in anticipation of the greatest wiring congestion, occurring in this region. Because prewired macro modules usually have a fixed cell-to-cell spacing, certain macros may not be placed in the center region (or the outer regions). TimberWolf allows the designation of cell locations as either suitable or unsuitable for a particular set of macro modules.

### 5.2 Gate Array Placement Algorithm

The TimberWolf gate array placement program can be used with either of two cost functions. The first cost function is based on the computation of net-crossing histograms for each horizontal and vertical channel of the placement region. The histograms are computed by considering the bounding box of each net and adding 1 to the histogram for each channel intersecting the bounding box. The sum of the histogram values for each horizontal and vertical channel is equivalent to summing the half perimeters of the bounding boxes of each net. Further, a net-crossing threshold value is assigned to each channel. If the number of nets crossing a channel exceeds the specified threshold value, a penalty is assessed proportional to the square of the number of net crossings exceeding the threshold. The threshold mechanism has the effect of evening

out the wiring congestion during the earlier stages of the annealing. This has shown to result in a lower value of the total wire length. A partitioning effect may be produced by setting the threshold of a particular channel to zero or a negative value. In this case, nets crossing this channel will be severely penalized.

The formulation of the cost function in terms of net-crossing histograms and threshold values was first introduced by Kirkpatrick, Gelatt, and Vecchi [1].

A second cost function for this program examines the local routing congestion more closely. For this cost function, each channel segment is assigned a threshold value. A channel segment is a portion of a horizontal or vertical channel with a length equal to the cell-center to cell-center spacing in that region of the array. For example, if the bounding box of a net encompasses 2 cells in the horizontal direction and 3 cells in the vertical direction, then a total of 17 segments are enclosed by the bounding box. The congestion per channel segment introduced by this net is approximated as the half perimeter of the bounding box (5) divided by the total number of segments enclosed (17). The factor of 5/17 is the estimated probability of occupancy for the given net in each of the 17 segments. The given net contributes zero to all other segments. The summation of the occupancy probabilities over all nets for a given segment is an estimate of the number of wiring tracks required. The cost function is then the sum of the expected occupancy of each segment plus a penalty assessed for each segment which has occupancy exceeding the corresponding threshold. Specifying a threshold value for each channel segment which reflects the actual fixed channel width increases the likelihood that the final placement will be routable. Furthermore, the total wire length will be minimized within the limits of these constraints.

### 5.3 Results

Experiments are currently being initiated on large gate array problems. To test the program and compare it with existing placement techniques, a set of standard benchmarks have been considered. These benchmarks are the ILLIAC IV computer boards reported by Stevens [5]. Note that the printed circuit board problem as stated for these examples is a particular case of the general gate array placement problem described in the previous subsection.

Wire length for a net was estimated by computing one half of the perimeter of the net's bounding box. The figure of merit is the sum of the estimated wire lengths for each net.

Three of the ILLIAC IV computer boards were tested. (1) The largest example required the placement of 151 modules on an 11 X 15 board. TimberWolf reduced the total wire length by 21% over Stevens' result and by 17% over the result published by Goto and Kuh [6]. (2) The second example required the placement of 108 modules on an 8 X 15 board. TimberWolf reduced the total wire length by 27% over the result published by Goto and Kuh. (3) The third example required the placement of 67 modules on a 5 X 15 board. TimberWolf reduced the total wire length by 17% over Stevens' result and 6% over the result published by Goto and Kuh.

The value of  $\alpha$  remained at a constant value of 0.90 for each of the examples. The results are summarized in Table 3. CPU times are for a VAX 11/780 running UNIX.

Circuit (#modules)	Stevens	Goto and Kuh	TimberWolf	CPU Time in Mins
151	2181	2098	1731	15
108	untested	1242	909	10
67	700	618	580	5

Table 3. Summary of Results  
TimberWolf Gate Array Placement Program

### 6. Macro/Custom Placement Optimization Program

#### 6.1 Introduction

This program optimizes the placement of macro cells and custom cells, as well as pads. The term macro cell will be used to refer to a cell contained in a cell library. That is, the dimensions of

the cell are known, as are the pin locations. The term *custom cell* will be used to refer to a block of circuitry known only to occupy an estimated area and to possess a list of pins.

The program places circuits comprised solely of macro cells as well as circuits comprised entirely of custom cells. Furthermore, the program will place circuits consisting of a combination of macro and custom cells. The macro cells and custom cells may be of any rectilinear shape.

TimberWolf allows the specification of lower and upper bounds for the aspect ratio of a custom cell. If a range of aspect ratios is given for a custom cell, TimberWolf will choose the shape of the cell which minimizes chip area.

Wire length calculations are based on the exact pin locations. Thus all possible orientations are considered for each cell.

Another feature of TimberWolf is the multiple region capability. This feature incorporates either a division of the chip into regions or the placement of multiple chips simultaneously. Interchanges of cells from different regions are permitted only if the regions belong to the same exchange class. The exchange class mechanism is extended to individual cells as well.

Pins are specified in several possible ways. (1) A pin may be given a particular fixed location. (2) A pin may be assigned to a particular side or sides of the cell. (3) A group of pins may be assigned to a particular side or sides of a cell. (4) A group of pins may be assigned to a particular sequence as well as a particular side or sides.

## 6.2 Macro/Custom Cell Placement Algorithm

The number of possible locations at which an *uncommitted* pin could be placed on a custom cell can often number into the thousands. Execution time considerations (as in the standard cell program) require that the pin locations be stored for each orientation of the cell. Clearly the amount of storage required can become excessively large. This potential problem is averted by defining a specified number of pin sites approximately evenly spaced along the periphery of a cell. Furthermore, each site is assigned a *capacity*. The capacity is a function of the number of pin locations encompassed by the site. During the annealing stages, pins are assigned to sites. Upon completion of the annealing algorithm, the pins for a given site are assigned to locations within the scope of the site based on the minimization of wire length. For accuracy considerations, the number of pin sites that are declared for a given placement problem is usually limited only by memory capacity.

The location of the pins on a macro cell are taken exactly. That is, their location is not approximated by the pin-site mechanism. The same is true for fixed-location pins on custom cells (if any are so specified). The capacity for a site in the vicinity of a fixed-location pin is correspondingly reduced.

The cost function consists of two independent parts. The first part is the total estimated wire length which is based on the sum over all nets of the half-perimeter of a net's bounding box. The second is the penalty function. The penalty function consists of two parts. (1) The first part is the sum of the overlap penalties for the cells. This penalty function was incorporated because of the usual difference in the size and shape of the cells. Often two cells are selected for interchange which differ in size and/or shape. Therefore, an exchange of location of these two cells often results in some overlap with one or more of the cells. Furthermore, the program often selects a single cell for a displacement to a new location or an aspect ratio change (in the case of custom cells). Once again, some overlap may result. The penalty assessed for an overlap of two cells is equal to the square of the quantity of the area of overlap plus an offset value. The offset parameter is selected to ensure that as the parameter  $T$  approaches zero, then the total overlap approaches zero. (2) The second part is the sum of the penalties assessed for the contents of a pin site exceeding its capacity. When a pin is displaced from an original site to a new site, the contents of the old site is reduced by 1 and the contents of the new site is increased by 1. The penalty assessed for a site is a product of the square of the amount by which the contents exceed the capacity, times a factor inversely related to the capacity of the site. This factor reflects the fact that exceeding the capacity by a given amount is a more serious violation for the sites with smaller capacities.

New states can be generated in several possible ways. (1) A pair of cells (either could be a macro cell or a custom cell) are selected for interchange. (2) A single cell is selected for a displacement to a new location. (3) A single cell is selected for an orientation change. (4) A custom cell is selected for an aspect ratio change. (5) An uncommitted pin (or sequence of pins) is assigned to a new site (or sites).

The ratio of single cell displacements to cell interchanges has a significant effect on the quality of the final placement. Initial experimental investigation has revealed that the best results are

obtained when the ratio is about 10 to 1.

The strategy for generating new states is based on the following: (1) A random number between one and the number of cells is generated. The cells are numbered sequentially from one. (2) A second random number is generated between 1 and the number of cells times 10. (3) If the two numbers both represent cells, then the pair of cells are interchanged to generate a new state. (4) If only the first number represents a cell, then the new state is generated by the displacement of the cell to a randomly selected location. If this new state was rejected, the next state generated is an orientation change for the cell. Similarly, if this new state was rejected and if the cell is a custom cell, then the next state generated is an aspect ratio change. Finally, if this new state was rejected, then a new state is generated by the selection of a uncommitted pin or group of uncommitted pins for transfer to a new pin site or sites.

In the latter stages of the algorithm, that is, when the value of  $T$  approaches zero, the displacement of a cell has very little chance of being accepted unless the displacement is very local. Similarly, an interchange of distant cells has a vanishingly small chance of being accepted. Thus a range limiter is employed which limits the range of the displacement of a cell or cells. Consequently, during the latter stages of the algorithm, the cells undergo many small displacements while gradually reducing wire length and forcing the penalty function to zero.

The TimberWolf macro/custom cell placement optimization program is currently being interfaced to CIPAR for testing purposes.

## 7. Conclusions

The TimberWolf placement and routing package has been shown to provide substantial chip area savings in comparison to existing standard cell layout programs. Substantial wire length reductions were also achieved for the gate array placement program for some benchmark examples. The TimberWolf macro/custom program, being tested now, is applicable to placement problems as complex as a multi-chip design employing a combination of macro cells and custom cells.

The TimberWolf package has demonstrated that the simulated annealing optimization technique is able to capture a wide range of user requirements. Our research group is also actively engaged in a theoretical investigation of the simulated annealing optimization technique.

The TimberWolf package will be interfaced to the SQUID database developed by our research group. In addition, the package will be interfaced with the YACR channel router, providing a complete standard cell placement and routing package.

The TimberWolf placement and routing package is written in the C programming language. The package currently runs under both the VAX/UNIX and VAX/VMS operating systems. The package is easily convertible to other systems supporting the C language.

## Acknowledgements

The authors would like to thank Jim Tobias of American Microsystems, Inc. for his support of this project and for his permission to interface TimberWolf to the CIPAR placement and routing package. The algorithmic part of this research has been sponsored by DARPA under grant number N00039-83-C-0107. The TimberWolf placement and routing package has been supported by a grant from MICRO. The authors wish to thank Fabio Romeo and Ken Keller for stimulating discussions. Carl Sechen wishes to thank Peter Moore, Tom Quarles, and Rick Spickemer for their significant contributions to his knowledge of the C programming language and the UNIX operating system.

## References

1. S. Kirkpatrick, C. Gelatt and M. Vecchi, Optimization by Simulated Annealing, IBM Computer Science/Engineering Technology Report, Watson Research Center, Yorktown Heights, NY, 1982.
2. M. Vecchi and S. Kirkpatrick, Global Wiring by Simulated Annealing, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* CAD-2, 4 (October 1983), 215-222.
3. J. Kruskal, On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, *Proc. Amer. Math. Soc.* 7, 1 (1956), 48-50.
4. M. Hannan, Net Wiring for Large Scale Integrated Circuits, IBM Research Report RC-1376, IBM, Feb. 1965.
5. J. Stevens, *Fast Heuristic Techniques for Placing and Wiring Printed Circuit Boards*, PhD Thesis, University of Illinois, Urbana, Illinois 1972.
6. S. Goto and E. Kuh, An Approach to the Two-Dimensional Placement Problem in Circuit Layout, *IEEE Trans. on Circuits and Systems* 26, 4 (April 1978), 208.

# Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools

JOHN K. OUSTERHOUT

**Abstract**—Corner stitching is a technique for representing rectangular two-dimensional objects. It is especially well suited for interactive VLSI layout editing systems. The data structure has two important features: first, empty space is represented explicitly; and second, rectangular areas are stitched together at their corners like a patchwork quilt. This organization results in fast algorithms (linear or constant expected time) for searching, creation, deletion, stretching, and compaction. The algorithms are presented under a simplified model of VLSI circuits, and the storage requirements of the structure are discussed. Corner stitching has been implemented in a working layout editor. Initial measurements indicate that it requires about three times as much memory space as the simplest possible representation.

## I. INTRODUCTION

**I**NTERACTIVE LAYOUT tools for integrated circuits place special burdens on their internal data structures. The data structures must be able to deal with large amounts of informa-

tion (one-half million or more geometrical elements in current layouts [7]) while providing instantaneous response to the designer. As the complexity of design increases, tools must give more and more powerful assistance to the designer in such areas as routing and validation. To support these intelligent tools, the underlying data structures must provide fast geometrical operations, such as locating neighbors for stretching and compaction, and locating empty space for routing. The data structures must also permit fast incremental modification so that they can be used in interactive systems.

Corner stitching is a data-structuring technique that meets these needs. As described here, it is limited to designs with Manhattan features (horizontal and vertical edges only); but within that framework it provides a variety of powerful operations, such as neighbor-finding, stretching, compaction, and channel-finding. The algorithms for the operations depend only on *local* information (the objects in the immediate vicinity of the operation). Their expected running times are generally linear in the number of nearby objects; in pathological cases (which are unlikely for actual layouts) the running times may be proportional to the overall design size or to the product of nearby objects and design size. Corner stitching is especially

Manuscript received January 5, 1983; revised June 20, 1983. This work was supported in part by the Defense Advanced Research Projects Agency (DoD), DARPA Order 3803, monitored by the Naval Electronic System Command under Contract N00039-81-K-0251.

The author is with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.



effective when the objects are relatively uniform in size, as is the case for low-level mask features. However, it also works well when there is variation in feature size. This occurs, for example, in a hierarchical layout where one cell might contain a few large subcells and many small wires to connect them together.

Corner stitching permits modifications to the database to be made quickly, since only local information is used in making the updates. Most existing systems that provide powerful operations such as routing and compaction do not provide inexpensive updates: small changes to the database can result in large amounts of recomputation. Corner stitching's combination of powerful operations and easy updates means that many powerful tools previously available only in "batch" mode can now be embedded in interactive systems.

## II. A SIMPLIFIED MODEL OF VLSI LAYOUTS

A VLSI layout is normally specified as a hierarchical collection of cells, where each cell contains geometrical shapes on several mask layers and pointers to subcells. As a convenience in presenting the data structure and algorithms, a simplified model is used in this paper. There is only a single mask layer, and hierarchy is ignored. For this paper, the author defines a "circuit" to be a collection of rectangles. There is a single design rule in the model: rectangles may not overlap. The simplified model makes it easier to present the data structure and algorithms. Section VII discusses how the simple model can be generalized to handle real VLSI layouts.

## III. EXISTING MECHANISMS

### 3.1. Linked Lists

The simplest possible technique for representing rectangles is just to keep all of them in a linked list. This technique is used in the Caesar system [6]: each cell is represented by a list of rectangles for each of the mask layers. Even though operations such as neighbor-finding require entire lists to be searched, the structure works well in Caesar for two reasons. First, large layouts are broken down hierarchically into many small cells; only the top-most cells in the hierarchy ever contain more than a few hundred rectangles or a few children [7]. Second, Caesar provides only very simple operations like painting and erasing. More complex functions such as design rule checking and compaction could not be implemented efficiently using rectangle lists.

### 3.2. Bins

The most popular data structures for VLSI are based on *bins* [2]. In bin-based systems, an imaginary square grid divides the area of the circuit into bins, as in Fig. 1. All of the rectangles intersecting a particular bin are linked together, and a two-dimensional array is used to locate the lists for different bins. Rectangles in a given area can be located quickly by indexing into the array and searching the (short) lists of relevant bins. The bin size is chosen as a tradeoff between time and space: as bins get larger, it takes longer to search the lists in each bin; as bins get smaller, rectangles begin to overlap several bins and hence occupy space on several lists.

Bin structures are most effective when rectangles have nearly

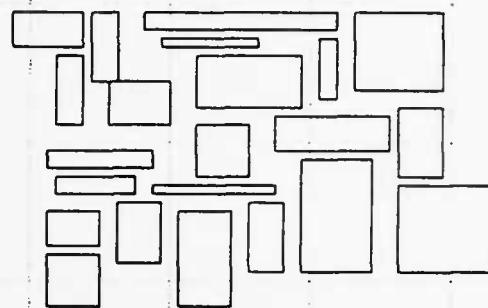


Fig. 1. In bin-based data structures, the circuit is divided by an imaginary grid, and all the rectangles intersecting a subarea are linked together.

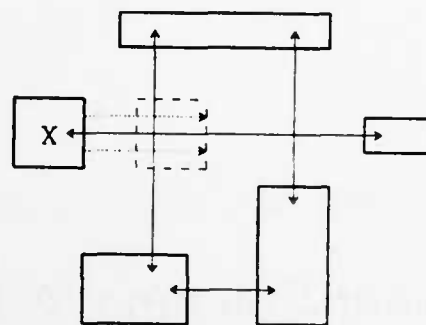


Fig. 2. Neighbor pointers can be used to indicate horizontal or vertical adjacency. However, if tile *X* is moved right, it is hard to update the vertical pointers without scanning the entire database.

uniform size and spatial distributions; they suffer from space and/or time inefficiencies when these conditions are not met. A pathological case is a cell with a few large child cells and many small rectangles to interconnect them. If bins are small, there will be many empty bins in the large areas of the subcells, resulting in wasted space for the bins; if bins are large, the bins in the wiring area will have many rectangles, resulting in slow searches. Hierarchical bin structures [4] have recently been proposed as a solution to the problems of nonuniformity. Although bins can be used to locate all the objects in an area, they do not directly embody the notion of *nearness*. To find the nearest object to a given one, it is necessary to search adjacent bins, working out from the object in a spiral fashion. Furthermore, bin structures do not indicate which areas of the chip are empty; empty areas must be reconstructed by scanning the bins. The need to constantly scan bins to recreate information makes bin structures clumsy at best, and inefficient at worst, especially for operations such as compaction and stretching.

### 3.3. Neighbor Pointers

A third class of data structures is based on neighbor pointers. In this technique, each rectangle contains pointers to rectangles that are adjacent to it in *x* and *y* (see Fig. 2). Neighbor pointers are a popular data structure for compaction programs such as Cabbage [3], since they provide information about relationships between objects. For example, a simple graph traversal can be used as part of compaction to determine the minimum feasible width of a cell.

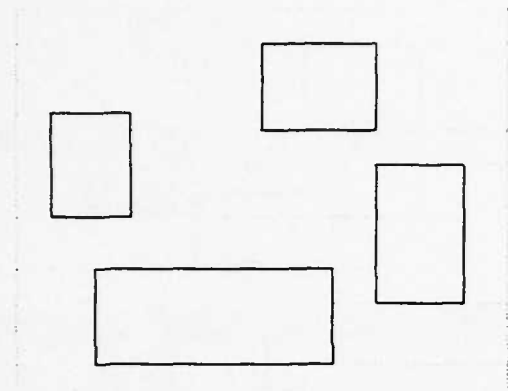


Fig. 3. An example of tiles in a corner-stitched data structure. Solid tiles are represented with dark lines, space tiles with dotted lines. The entire area of the circuit is covered with tiles. Space tiles are made as wide as possible.

Neighbor pointers have two drawbacks. First, modifications to the structure generally require all the pointers to be recomputed. For example, if an object is moved horizontally, as in Fig. 2, vertical pointers may be invalidated. There is no simple way to correct the vertical pointers short of scanning the entire database. The second problem with neighbor pointers is that they provide no assistance in locating empty space for routing, since only the occupied space is represented explicitly. For these two reasons, neighbor pointers do not appear to be well-suited to interactive systems or those that provide routing aids.

#### IV. CORNER STITCHING

Corner stitching arose from a consideration of the weaknesses of the above mechanisms, and has two features that distinguish it from them. The first important feature is that all space, both empty and occupied, is represented explicitly in the database. The second feature is a novel way of linking together the objects at their corners. These *corner stitches* permit easy modification of the database, and lead to efficient implementations for a variety of operations.

Fig. 3 shows four objects represented in the corner stitching scheme. The picture resembles a mosaic with rectangular tiles of two types, space and solid. The tiles must be rectangles with sides parallel to the axes. Tiles contain their lower and left edges, but not their upper or right edges, so every point in the plane is present in exactly one tile. The entire plane is covered from  $-\infty$  to  $+\infty$  in both  $x$  and  $y$  (in practice, the largest representable positive and negative numbers are used for the infinities). Coverage to infinity is achieved by extending the outermost space tiles; no extra tiles are required.

The space tiles are organized as *maximal horizontal strips*. This means that no space tile has other space tiles immediately to its right or left. When modifying the database, horizontally adjacent space tiles must be split into shorter tiles and then joined into maximal strips, as shown in Fig. 4. After making sure that space tiles are as wide as possible, vertically adjacent tiles are merged together if they have the same horizontal span. The representation of space is of no consequence to the VLSI layout or to the designer, and will not even be visible in real systems. However, the maximal horizontal strip representation is crucial to the space and time efficiency of the tools, as we

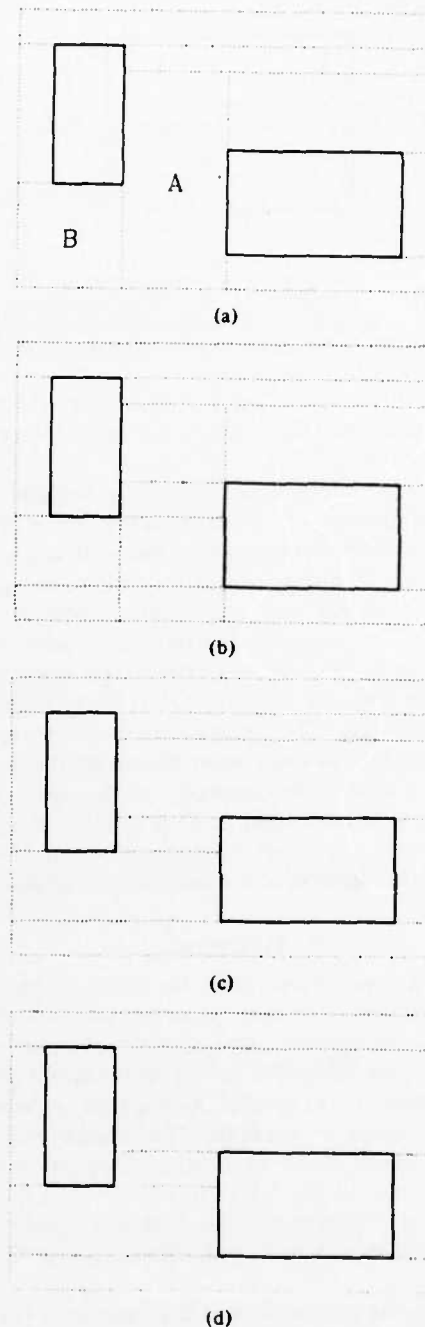


Fig. 4. No space tile may have another space tile to its immediate right or left. In this example, tiles A and B in (a) must be split into the shorter tiles of (b), then merged together into wide strips in (c), and finally merged vertically in (d).

shall see in Sections V and VI. Among its other properties, the horizontal-strip representation is unique: there is one and only one decomposition of space for each arrangement of solid tiles.

Tiles are linked by a set of pointers at their corners, called *corner stitches*. Each tile contains four stitches, two at its lower-left corner and two at its upper right corner, as illustrated in Fig. 5. Since there is one pointer in each of the four directions, the stitches provide a form of sorting that is equivalent to neighbor pointers. Originally, eight stitches were used, two

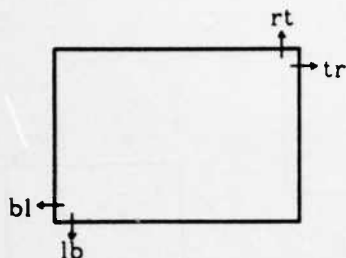


Fig. 5. Each tile is connected to its neighbors by four pointers called corner stitches. The names of the stitches indicate the tiles they point to: the *tr* stitch points to the tile's topmost right neighbor, the *lb* stitch points to the tile's leftmost bottom neighbor, and so on.

at each of the four corners, but four turned out to be sufficient for the algorithms presented here. The choice of these particular four stitches is important.

The tile/stitch representation has several attractive features, which will be illustrated in the sections that follow. First, the mechanism combines both horizontal and vertical pointers in a single structure. The space tiles provide a form of registration between the horizontal and vertical information and make it easy to keep all the pointers up to date as the circuit is modified. Because the space tiles may vary in size (as opposed to fixed-size bins), the structure adapts naturally to variations in the sizes of the solid tiles. The maximal horizontal strip representation of space results in clean upper bounds on the number of space tiles and also on the complexity of the algorithms. All tiles have the same number of pointers to other tiles, so they occupy the same number of bytes of storage; this simplifies the database management and reduces the "constant factors" in algorithms.

## V. ALGORITHMS

This section presents algorithms for manipulating the tiles and corner stitches. The most important attribute of all the algorithms is their locality: each algorithm depends only on information in the immediate vicinity of the operation. None of the algorithms has an expected running time any worse than linear in the number of tiles in the affected area. Pathological cases will be shown where the algorithms require time linear, or even quadratic, in the overall layout size, but in practice (particularly for VLSI layouts, which tend to be densely packed) their running times are small and independent of the size of the layout.

In discussing the performance of the algorithms, the corner stitches provide a good unit of measure. The complexity of the algorithms will be discussed in terms of the number of stitches that must be traversed (or, alternatively, the number of tiles that must be visited) and/or the number of stitches that must be modified.

### 5.1. Point Finding

Several different kinds of searching are facilitated by corner stitching. One of the most common operations is to find the tile at a given  $(x, y)$  location. Fig. 6 illustrates how this can be done with corner stitching. The algorithm iterates in  $x$  and  $y$ , starting from any given tile in the database:

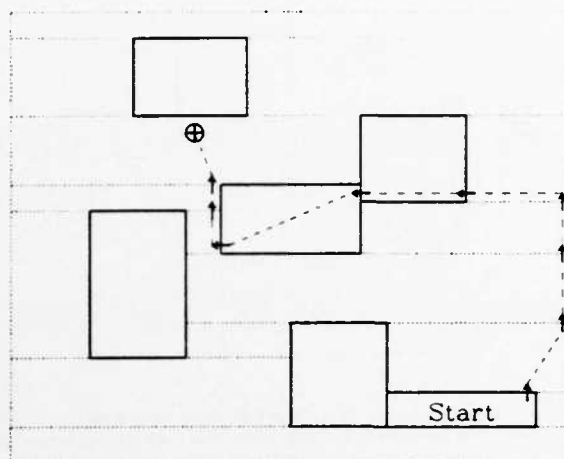


Fig. 6. To locate the tile containing a given point, alternate between up/down and left/right motions.

1) First move up or down, using right top (*rt*) and left bottom (*lb*) stitches, until a tile is found whose vertical range contains the desired point.

2) Then move left or right, using *tr* and *lb* stitches, until a tile is found whose horizontal range contains the desired point.

3) Since the horizontal motion may have introduced a vertical misalignment, steps 1) and 2) may have to be iterated several times to locate the tile containing the point. The convexity of the tiles guarantees that the algorithm will converge.

In the worst case, this algorithm may require every tile in the entire structure to be searched (this happens, for example, if all the tiles in the structure are in a single column or row). Fortunately, the average case behavior is much better than this. If there are a total of  $N$  space or solid tiles and they are of relatively uniform size, then on the order of  $\sqrt{N}$  tiles will be passed through in the average case. For a layout containing a million tiles (which is typical of the fully expanded mask sets of current VLSI circuits), this means a few thousand tiles will have to be touched.

In interactive systems, there is a simple way to reduce the time spent in point finding: keep a pointer around to any tile in the approximate area where the designer is working. When a large design is being edited, the designer's attention is generally focused on a small piece of the design (e.g., a piece that can be viewed comfortably on a graphic device). If a *hint* tile in this area is remembered for reference, the search time depends only on how much is on the screen, not how large the design is.

The point-finding algorithm illustrates a general feature of most of the algorithms: misalignment. While searching horizontally, it is possible to lose the vertical alignment, so the algorithm must iterate over horizontal and vertical motions. See Fig. 6 for an example. In general, large tiles can cause the algorithms of this paper to wander arbitrarily far outside their areas of interest. When this happens, the algorithms must traverse stitches to get back to the desired area again. Extreme misalignment results in worst-case behavior for many of the

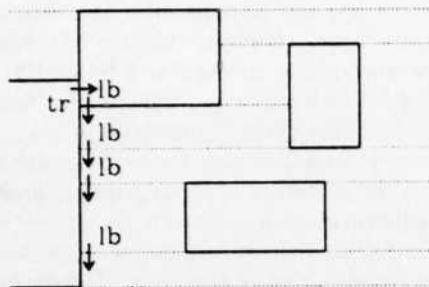


Fig. 7. The corner stitches provide a simple way to find all the tiles that touch one side of a given tile.

algorithms. Fortunately, severe misalignment is unlikely for densely packed designs.

### 5.2. Neighbor Finding

Another common searching operation is neighbor finding: find all the tiles that touch one side of a given tile. Neighbor finding is useful for design rule checking, compaction, circuit extraction, and tracing out connected nets. Fig. 7 illustrates how to find all the tiles that touch the right side of a given tile:

- 1) Follow the *tr* stitch of the starting tile to find its topmost right neighbor.
- 2) Then trace down through *lb* stitches until all the neighbors have been found (the last neighbor is the first tile encountered whose lower *y* coordinate is less than or equal to the lower *y* coordinate of the starting tile).

Similar algorithms can be devised to search each of the other sides. The time for the search is linear in the number of neighbors. As shown in Appendix 1, the expected number of neighbors is one or two along each side. In layouts where tile sizes vary greatly, the number of neighbors will, on average, be proportional to the length of the side.

### 5.3. Area Searches

A third form of searching is to see if there are any solid tiles within a given area. This can be accomplished in the following manner using corner stitches (see Fig. 8):

- 1) Use the point-finding algorithm to locate the tile containing the upper left corner of the area of interest.
- 2) See if the tile is solid. If not, it must be a space tile. See if its right edge is within the area of interest. If so, it is the edge of a solid tile.
- 3) If a solid tile was found in step 2), then the search is complete. If no solid tile was found, then move down to the next tile touching the right edge of the area of interest. This can be done either by invoking the point-finding algorithm, or by traversing the *lb* stitch down and then traversing *tr* stitches right until the desired tile is found.
- 4) Repeat steps 2) and 3) until either a solid tile is found or the bottom of the area of interest is reached.

As with the other operations, the time necessary for this

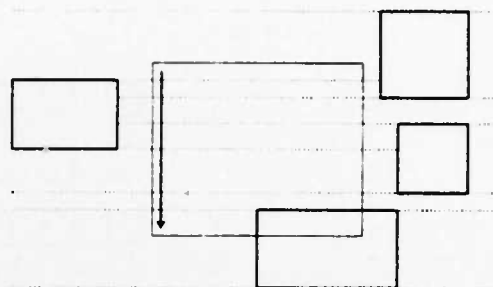


Fig. 8. To search a rectangular area for a solid tile, work down along the left edge of the area. Each tile along the edge must be either a solid tile, a space tile that spans the entire area, or a space tile with a solid tile just to its right.

operation depends only on local features: the number of tiles in and around the area of interest. The cost can be measured by counting the number of stitches that must be traversed. The number of iterations through the algorithm will be proportional to the height of the area (assuming, as always, a relatively uniform size distribution). In each iteration, it may be necessary to traverse one stitch in step 2). In addition, step 3) will cause a misalignment of about 1/2 tile in the average case. Thus the total running time is linear in the height of the search area, and does not depend at all on the width of the search area. In worst-case situations like the one shown in Fig. 9(a), misalignments could cause the running time to be proportional to the total number of tiles in the layout.

### 5.4. Directed Area Enumeration

The algorithm in Section 5.3 determines if there are any solid tiles in an area. However, for many applications, such as compaction and layout rule checking, it is useful to enumerate *all* the tiles in a given area, i.e., to "visit" each tile exactly once. Furthermore, it is often useful to do this in a particular direction. For example, during a left-to-right compaction, it is important that a tile not be processed until all the tiles on its left have been processed. This section presents an algorithm wherein each tile is visited only after all the tiles above it and to its left have been visited. I call such an enumeration a *directed enumeration*. Corner stitching makes this a linear time operation. Fig. 10 shows the enumeration order for an example case.

- 1) As for the area-searching algorithm, use the point-finding algorithm to locate the tile at the top left corner of the area of interest. Then step down through all the tiles along the left edge, using the same technique as in area searching.
- 2) For each tile found in step 1), enumerate it recursively using the R procedure given in lines R1) through R5).
  - R1) Enumerate the tile (this will generally involve some application-specific processing).
  - R2) If the right edge of the tile is outside of the search area, then return from the R procedure.
  - R3) Otherwise, use the neighbor-finding algorithm to locate all the tiles that touch the right side of the current tile and also intersect the search area.



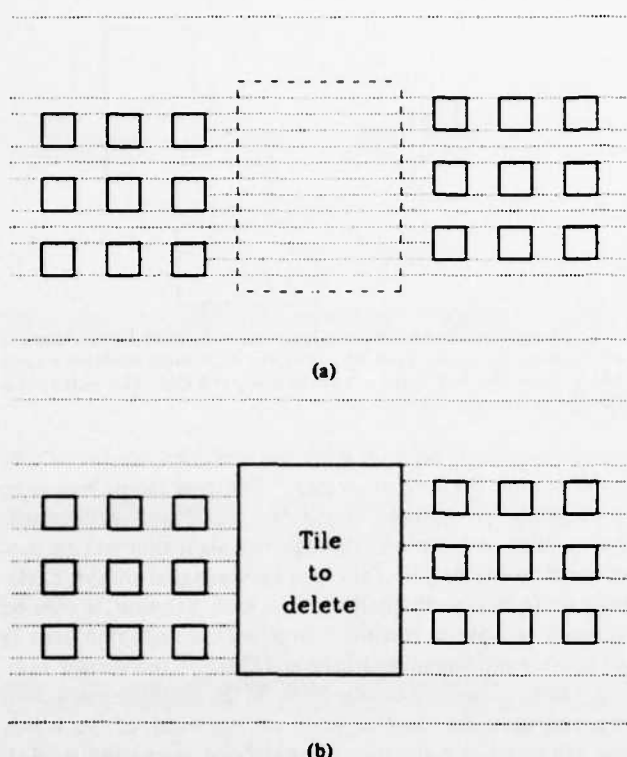


Fig. 9. Two pathological structures. In (a), area searches of the dashed area are slow because of severe misalignment during step 3) of the algorithm. It is also slow to create a tile in the dashed area at (a), which produces the situation in (b), or delete the labeled tile in (b) to get back the situation in (a): when splitting and merging space tiles, corner stitches must be modified in every solid tile in the circuit.

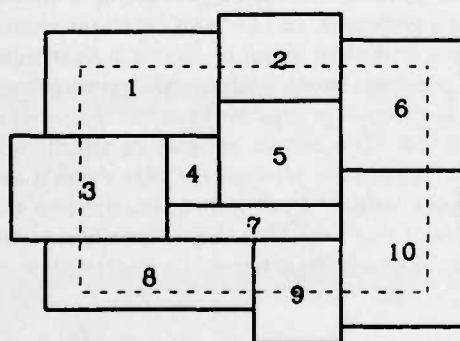


Fig. 10. An example of directed enumeration. When doing an upper left to lower right enumeration of the dashed area, the tiles will be visited in order of their numbers.

R4) For each of these neighbors, if the bottom left corner of the neighbor touches the current tile then call R to enumerate the neighbor recursively (for example, this occurs in Fig. 10 when tile 1 is the current tile and tile 2 is the neighbor).

R5) Or, if the bottom edge of the search area cuts both the current tile and the neighbor, then call R to enumerate the neighbor recursively (in Fig. 10, this occurs when tile 8 is the current tile and tile 9 is the neighbor).

The expected running time of the directed enumeration algorithm is linear in the number of tiles intersecting the search area. This can be shown by the following arguments. The

checks in steps R4) and R5) guarantee that each tile is enumerated exactly once. However, a tile may be checked several times before satisfying the checks in step R4) or R5): it will be checked once for each tile that touches its left side. The total expected running time of the algorithm is thus proportional to the total number of adjacencies within the search area. Appendix I uses the properties of planar graphs to prove that the number of adjacencies must be linear in the number of tiles.

In the worst case, directed area enumeration could require every tile in the circuit to be examined. This happens if tiles stick out far above the top edge of the area being enumerated: all of their neighbors must be enumerated in step R3), even though most of them do not intersect the area of interest.

The algorithm for directed enumeration does not depend on the fact that space tiles are maximal horizontal strips. In fact, it does not even distinguish between solid and space tiles. A similar algorithm can be devised to reverse the direction of enumeration (from lower right to upper left). But, it is much more difficult to recode the algorithm to operate from lower left to upper right, or from upper right to lower left (this is because there are no corner stitches emanating from the lower right or upper left corners of tiles).

### 5.5 Tile Creation

The first step in creating a new solid tile is to check to see that there are no existing solid tiles in the desired area of the new tile. The area-search algorithm can check this. The second step is to insert the tile into the data structure, clipping and merging space tiles and updating corner stitches as shown in Fig. 11. The insertion algorithm is as follows:

- 1) Find the space tile containing the top edge of the area to be occupied by the new tile (because of the strip property, a single space tile must contain the entire edge).

- 2) Split the top space tile along a horizontal line into a piece entirely above the new tile and a piece overlapping the new tile. Update corner stitches in the tiles adjoining the new tile.

- 3) Find the space tile containing the bottom edge of the new solid tile, split it in the same fashion, and update stitches around it.

- 4) Work down along the left side of the area of the new tile, as for the area-search algorithm. Each tile along this edge must be a space tile that spans the entire width of the new solid tile. Split the space tile into a piece entirely to the left of the new tile, a piece entirely to the right of the new tile, and a piece entirely within the new tile. This splitting may make it possible to merge the left and right remainders vertically with the tiles just above them: merge whenever possible. Finally, merge the center space tile with the solid tile that is forming. Each split or merge requires stitches to be updated in adjoining tiles.

The speed of the creation algorithm is determined by the cost of splitting and merging the space tiles that cross the area. The number of space tiles depends on the number of solid tiles in the left and right shadows of the new tile. One can devise cases where the number of space tiles is arbitrarily high, but in practice, the expected number is proportional to the relative height of the new tile in comparison to the tiles around it. Appendix B discusses the cost of splitting and merging tiles. In the average case it is constant; for very large tiles it is proportional to the circumference of the tile. This means

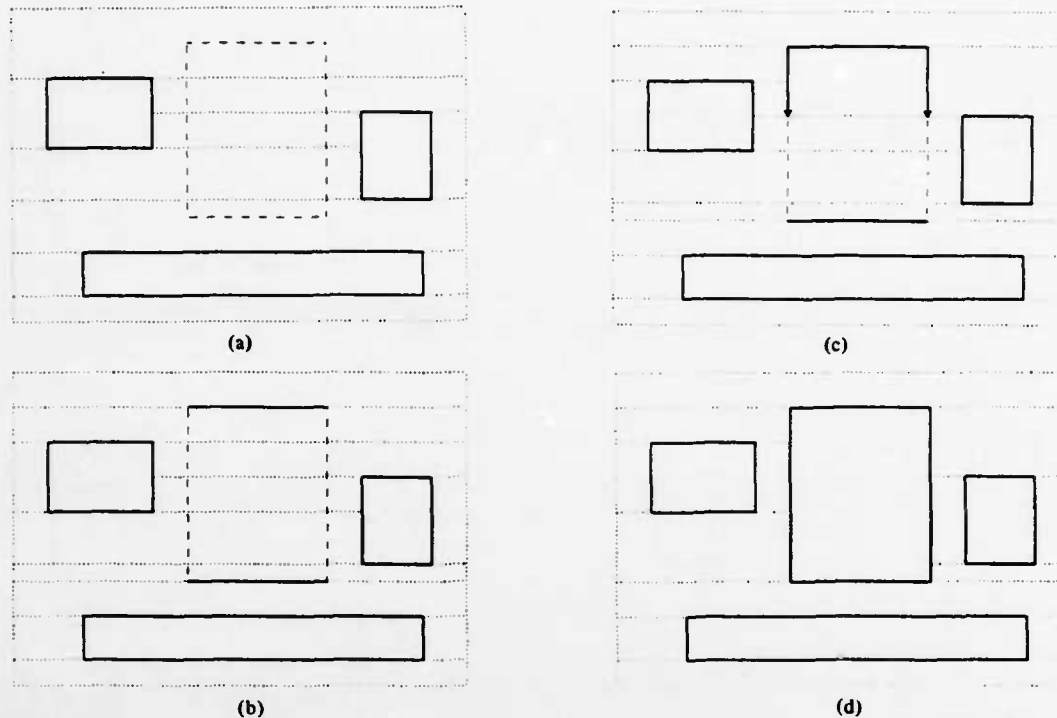


Fig. 11. Inserting a new solid tile into the data structure. (a) shows the desired location of the new tile. In (b) the space tiles containing the top and bottom edges of the new solid tile are split. In (c) and (d) the area of the new tile is traversed from top to bottom, splitting and joining space tiles on either side and pointing their stitches at the new solid tile.

that in the worst possible case, the cost of creating a new tile could be proportional to the total number of tiles in the layout (see Fig. 9(a)). In the average case, the running time is constant if the new tile is about the same size as the tiles around it; if the new tile is much larger than its neighbors, then the running time is proportional to the height of the new tile and independent of its width.

### 5.6 Tile Deletion

Tile deletion is complicated by the need to split and merge space tiles so as to maintain the horizontal-strip representation. The algorithm below works in a mostly clockwise fashion around the tile being deleted, which is referred to as the *dead tile*. See Fig. 12 for an example.

- 1) Change the type of the dead tile to "space".
- 2) Use the neighbor-finding algorithm to search from top to bottom through all the tiles that adjoin the right edge of the dead tile.
- 3) For each space tile found in step 2), split either the neighbor or the dead tile, or both, so that the two tiles have the same vertical span, then merge the tiles together horizontally.
- 4) When the bottom edge of the original dead tile is reached, scan upwards along the left edge of the original dead tile to find all the space tiles that are left neighbors of the original dead tile.
- 5) For each space tile found in step 4), merge the space tile with the adjoining remains of the original dead tile. Do this by repeating steps 2)-3), treating the current space tile like the dead tile in steps 2)-3).
- 6) It is also necessary to do vertical merging in step 5). After each horizontal merge in step 5), check to see if the re-

sult tile can be merged with the tiles just above and below it, and merge if possible.

As with the other algorithms, deletion could require a great deal of time in pathological cases. For example, Fig. 9(b) shows a situation where corner stitches will have to be examined and modified in every single tile in the layout, so running time will be proportional to the overall layout size. However, situations like this are not likely in integrated circuits. If the tiles are roughly uniform in size and distribution, then the number of splits and joins will be constant and the running time will also be constant. When a large tile is being deleted, the running time will be proportional to the number of left and right neighbors of the tile, which is proportional to the tile's height.

### 5.7 Plowing

Plowing is an example of an important operation that cannot easily be implemented with most existing data structures. When one piece of a large design is moved, it is often desirable for other pieces of the design lying in the path of motion to move as well, as if the original piece were a plow. Ideally, such a motion will stretch or shrink the design while maintaining design rules and connectivity. Plowing can be accomplished with corner stitching in the following way:

- 1) Determine the rectangular area that will be swept out by the motion of the original tile (see Fig. 13).
- 2) Use the area-finding algorithm to see if there are any solid tiles in the plow area. If a solid tile is found, invoke the plow algorithm recursively to move the tile out of the plow area. Repeat this step until no solid tiles are found.
- 3) Delete the original tile from its old location and create it at the new position.

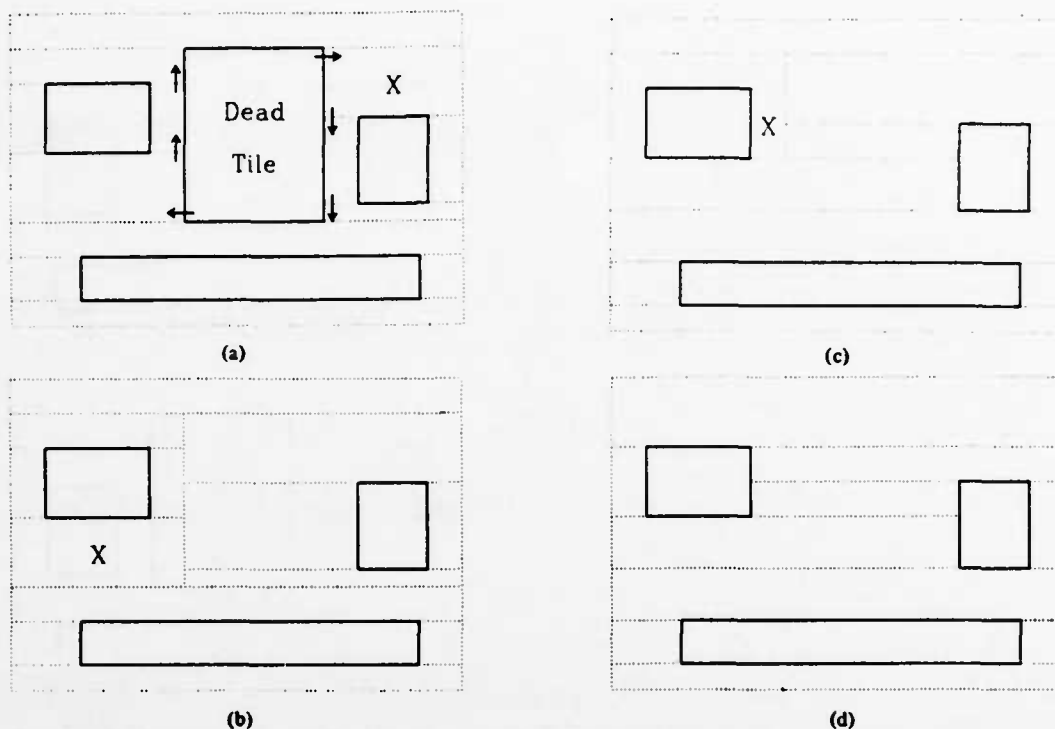


Fig. 12. An example of tile deletion. In each figure, tile *X* is the next one to be processed. (a) Shows the initial tile arrangement and the clockwise order in which stitches will be traversed around the dead tile to merge it with adjacent space tiles. In (b) the downward sweep along the right edge has been completed (note that the left edge of the dead tile is still intact). In (c) the upward sweep along the left edge is partially complete, and (d) shows the final situation.

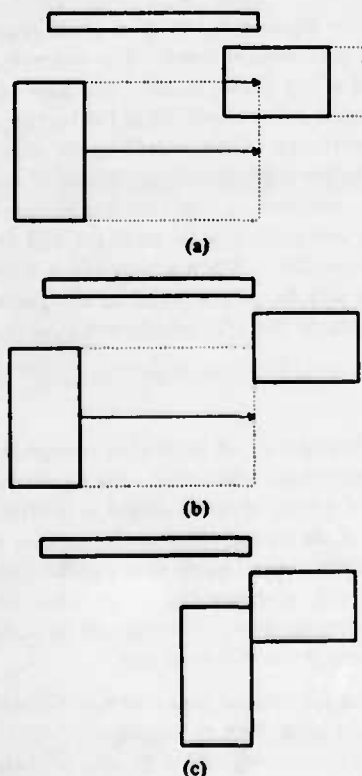


Fig. 13. An example of plowing: (a) determine the area to be swept out by the motion; (b) recursively move all solid tiles out of this area; and (c) move the original tile.

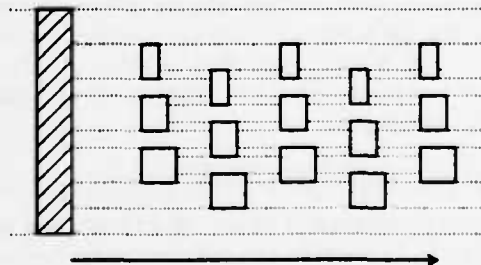


Fig. 14. Using a top-to-bottom area search with the simple plowing algorithm, this structure will cause the rightmost tiles to be moved many times when the cross-hatched tile is plowed to the right. Total running time will be exponential in the circuit size.

Unfortunately, this simple algorithm suffers from terrible worst-case behavior. Lattice structures like the one in Fig. 14 can require up to  $2^N$  recursive tile moves to clear  $N$  tiles out of the plow area. It seems likely that structures similar to the one in Fig. 14 may occur in actual circuits. Fortunately, the algorithm can be made to run in linear expected time by ordering the recursive processing so that a tile is not moved until its final position is known (i.e., it is not processed until all the tiles that can affect its final position have been processed). The code is somewhat complex, and is different for horizontal plowing than for vertical plowing. Appendixes C and D develop the linear time algorithm in detail. In the worst case, the algorithms of Appendixes C and D could require  $MN$  time, where  $M$  is the total number of tiles that have to be moved



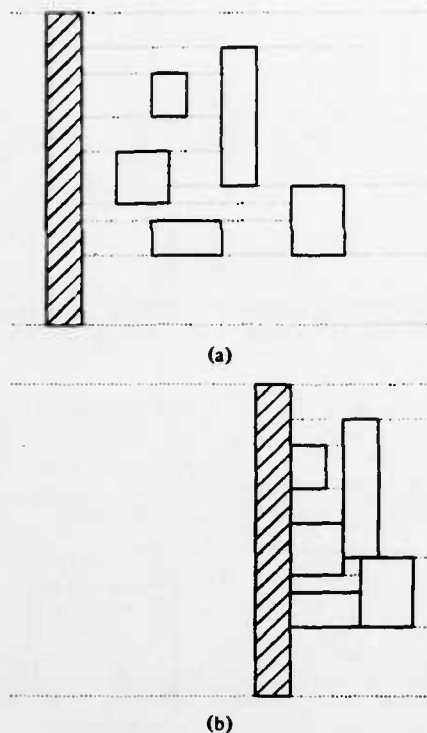


Fig. 15. To compact a layout horizontally, plow a large additional tile (cross-hatched in the figure) across the layout: (a) shows the configuration before the plow, and (b) shows the compacted configuration afterwards. The tile acts like a broom and compacts as it sweeps.

and  $N$  is the size of the circuit. In the average case they require time linear in  $M$ .

### 5.8 Compaction

Most existing algorithms for compaction require  $N^2$  time in the worst case for a layout containing  $N$  elements, and have been empirically observed to have average running time close to  $N^{1.2}$  [8]. With corner stitching, compaction is linear in the size of the layout. Compaction in a single direction can be achieved in a simple way by plowing a large tile across the layout, as shown in Fig. 15. The linear expected time for plowing results in linear expected time for compaction. The worst-case compaction time is still  $N^2$  using corner stitching.

There are two keys to the speed of compaction in corner stitching. The first, and most important, is that all the dependencies between tiles are maintained dynamically. In other compaction systems, the dependencies must be reconstructed after each change to the layout; the algorithms for generating dependencies limit the overall speed of compaction. The second key is that the layout is planar. This means that the number of adjacencies is linear in the number of tiles, and hence, the whole layout can be scanned in time proportional to the number of tiles.

### 5.9 Channel Finding

Channel information is constantly available in the form of the space tiles. The corner stitches make it possible to find connected channels and thereby trace out signal paths. Of

TABLE I

Corner stitching requires about 40 percent more storage per tile than linked list systems like Caesar. Only the lower and left coordinates of each tile need be stored in corner stitching, since the upper and right coordinates can be gotten by examining the lower and left coordinates of neighboring tiles.

	Caesar	Corner Stitching
Coordinates	$x_1, y_1, x_2, y_2$ (16 bytes)	$x_1, y_1$ (8 bytes)
Pointers	1 link (4 bytes)	4 stitches (16 bytes)
Tile Type	not needed	(4 bytes)
Total	20 bytes	28 bytes

course, some routers may prefer a different representation of channels than maximal horizontal strips; if this is the case, then conversion will be necessary to cast the space tiles into a form suitable for routing.

## VI. SPACE REQUIREMENTS

Because of the enormous size of VLSI designs, a data structure used for VLSI CAD must be space efficient if it is to be effective. For example, the hierarchical representation of a 45 000-transistor chip requires about  $1.5 \times 10^6$  bytes of main memory in Caesar. Corner stitching requires more information to be kept in the data structure than systems like Caesar. Table I compares corner stitching to the linked-list scheme of Caesar. Corner stitching requires three more pointers than Caesar, plus a type field (in linked-list systems all the tiles on a given list are of the same type). Corner stitching saves space by storing only the lower and left coordinates of each tile, instead of four coordinates: the upper and right coordinates of a tile can be gotten from the lower and left coordinates of neighboring tiles. As a result, corner-stitched tiles are about 40 percent larger than Caesar tiles. In addition, there are many more tiles in corner stitching than in other systems since corner stitching requires empty space to be represented. If there are many space tiles, then corner stitching will require too much space to be practical. Furthermore, most of the algorithms depend on the total number of tiles in an area, including both space and solid tiles; if there are many space tiles, the algorithms will be inefficient.

In a circuit with  $N$  solid tiles, there will never be more than  $3N + 1$  space tiles. Furthermore, the horizontal-strip representation is at least as efficient (in the worst case) as any other rectangle-based representation of space. In actual circuit layouts, the number of space tiles is about equal to the number of solid tiles.

The proof of the  $3N + 1$  upper limit is due to C. Séquin. To see that no more than  $3N + 1$  space tiles are needed for  $N$  solid tiles, place the solid tiles one at a time in order from right to left as shown in Fig. 16. Initially there is a single space tile. When each solid tile is placed, it can result in no more than three new space tiles: the top and bottom edges may each cause a space tile to be split, and a new space tile will be created in the shadow to the left of the solid tile. Because we place the solid tiles in order, there can be no solid tiles in the shadow. This means that only a single space tile will be created there. Although the solid tiles were placed in a particular order to demonstrate the  $3N + 1$  limit, the final configuration is independent of the order in which the tiles

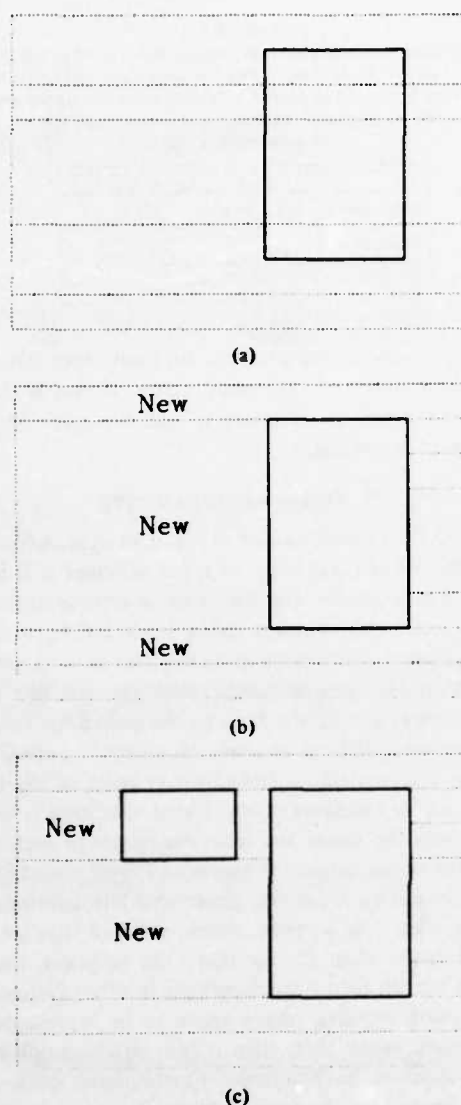


Fig. 16. Both (a) and (b) show that if solid tiles are inserted in order from right to left, each tile causes no more than three additional space tiles to be created. However, if edges of the new tile align with edges of old tiles, as in (c), less than three additional space tiles will be required.

are placed (the horizontal-strip property guarantees this). Thus the result is valid regardless of the order of solid-tile creation.

There are many other ways to organize space tiles besides maximal horizontal strips. However, in the worst case, no representation of space can use less than  $3N + 1$  space tiles. This worst case occurs when no two solid tiles have colinear edges. Fig. 17 shows one such situation.

Substantially fewer than  $3N + 1$  space tiles are needed for actual VLSI applications. Fewer space tiles are needed whenever edges of neighboring solid tiles align. For example, Fig. 16(c) shows a situation where the placement of a solid tile only adds two space tiles instead of three. In integrated circuits, the solid tiles must touch each other to achieve electrical connectivity, so the number of space tiles actually needed is much less than  $3N$ . Table II shows sample data

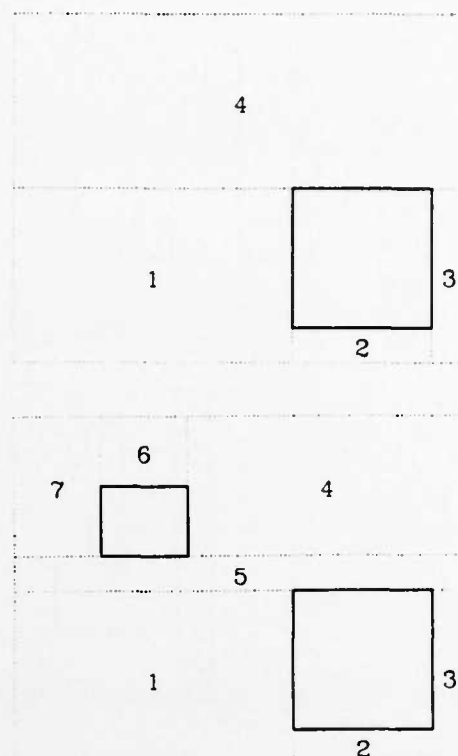


Fig. 17. In pathological situations where no two solid tiles have colinear edges, at least  $3N + 1$  tiles must be used to represent space, regardless of whether or not horizontal strips are used.

TABLE II

For actual layouts, corner stitching requires about one space tile for each solid tile. The first case consists of all the global routing for the RISC I microprocessor (i.e., all the rectangles in the topmost cell of the hierarchy). The routing is sparse. The second and third cases consist of cells of another microprocessor under development.

Circuit	Solid Tiles	Space Tiles	Space/Solid
Global Routing	8037	8473	1.05
ALU Latch	177	174	.98
Register Cell	77	65	.84

gathered from three cells using a layout editor based on corner stitching. On the average, about one space tile is required for each solid tile. This means that the total storage required for geometry in corner stitching will be between two and a half and three times as great as in systems like Caesar. This result applies even when the mask layers are sparse, as in the global routing example.

## VII. USING CORNER STITCHING FOR REAL VLSI

The scheme presented here must be extended in several ways to make it practical for real integrated circuits. This section presents some of the important issues and discusses possible solutions. To date, there have been two implementations of corner stitching. A toy implementation was built using exactly the model and algorithms of this paper, in order to test the basic viability of the ideas. About 1100 lines of C code were required to implement all the algorithms, including compaction, and for small test cases (100 tiles) response was instantaneous for all operations.

As a result of the successful toy implementation, we have undertaken the development of a full-fledged VLSI layout editor based on corner stitching. It has just recently become operational. Although stretching and compaction have not been implemented yet, the current system is at least as powerful as its predecessor, Caesar, and is being used by chip designers at the University of California at Berkeley.

The first generalization of the simple scheme is to provide for multiple mask layers. There are several ways to accomplish this. One alternative is to permit many different types of solid tiles, one type for each possible combination of mask layers. Unfortunately, this scheme will result in enormous numbers of tiny tiles in places where several mask layers cross each other. Many of the layer crossings are not relevant, so the fragmentation of the tile structure wastes space unnecessarily (for example, it doesn't matter where metal crosses polysilicon or diffusion, unless there are contact cuts present). Another alternative is to keep a separate corner-stitched "plane" for each mask layer. This scheme will be relatively space efficient, but will require frequent cross-registration between planes during operations such as plowing and design-rule checking that deal with layer interactions.

For our layout editor based on corner stitching, we used a combination of these two schemes. The polysilicon, diffusion, and implant layers are kept together in a single corner-stitched plane with different types of solid tiles for each layer combination. This makes sense because most of the different combinations of these layers are distinct electrically. Each metal layer is kept in its own corner-stitched plane, since they interact only weakly with each other and with the rest of the circuit. Because contacts provide a connection between layers, they are duplicated in each of the planes that they connect. Under this scheme, the corner-stitched representation corresponds almost exactly to the electrical circuit, since the transistors (combinations of polysilicon and diffusion and implants) are represented by special tile types. Furthermore, this particular division of mask layers among planes allows each plane to be design-rule checked independently.

To handle hierarchical designs, our layout editor keeps a separate set of tile planes for each cell in the design. An additional corner-stitched plane per cell is used to keep track of the cell's subcells. A different tile type is used in this plane for each distinct subcell or overlap area between subcells.

Design-rule checking is trivial in the simple model. The only design rule is that there can be no solid-tile overlap; this condition is enforced by the creation and plowing routines. In actual IC designs, the design rules will include more complex spacing and separation rules that are different for different tile types. For the corner-stitched editor, we have implemented a simple design-rule checker similar to Lyra [1] except that it is edge based instead of corner based. It scans a corner-stitched plane, generates constraints at each edge based on the tile types on either side of the edge, and uses area enumeration to check the constraints. To handle areas of overlap between subcells, the design-rule checker extracts information from the separate planes of the subcells into an auxiliary corner-stitched structure and then checks the auxiliary structure.

The plow algorithm is also affected by more complex design

TABLE III

TYPICAL AND WORST-CASE RUNNING TIMES FOR THE ALGORITHMS.  $M$  refers to the number of tiles of direct interest to the algorithm (e.g., the number of tiles being enumerated in area enumeration, or the number of tiles removed in plowing).  $N$  refers to the total number of tiles in the circuit.

Algorithm	Expected Time	Worst-case Time
Point Search	$\sqrt{N}$	$N$
Point Search (with hint)	constant	$N$
Neighbor Search	$M$	$M$
Area Search	$M$	$N$
Directed Area Enumeration	$M$	$N$
Tile Creation	constant	$N$
Tile Deletion	constant	$N$
Plowing	$M$	$MN$
Compaction	$N$	$N^2$

rules, and must deal with connectivity as well. Although the implementation of plowing is not yet complete, real VLSI design rules appear to be accommodated by selectively expanding the plow area to maintain proper spacings. For example, if the metal-metal spacing must be three units, then, when plowing a metal tile, all unrelated metal must be cleared from an area three units larger on all sides than the area swept out by the tile's motion. Connectivity appears to be handled by selectively stretching or shrinking some tiles, rather than moving them.

In some industrial environments, the Manhattan restriction may be intolerable. Where this is the case, it may be possible to accommodate 45°-angles by using trapezoids instead of rectangles. Degenerate trapezoids can be used to represent triangles. We do not plan to implement non-Manhattan features in our system, since in our environment, the Manhattan restriction is acceptable (and even desirable, since it tends to simplify designs and make tools run two to ten times faster). The Manhattan design style seems to be gaining more and more acceptance in the integrated circuit design community as a whole. For example, the Caesar editor, which is Manhattan, is now being used at nearly 200 industrial and university sites.

## VIII. CONCLUSION

Corner stitching is a powerful technique for representing geometrical data. Its two most important features are a) it represents empty space explicitly, and b) it links together tiles of various types at their corners. These two features make it possible to implement a variety of important operations that operate purely locally. The efficiency of the algorithms depends only on local information and not on the overall circuit size. The database can be modified incrementally, so that one portion of the design can be changed without invalidating the pointer information of any other piece of the design. Corner stitching is effective both for densely packed circuits and for sparse ones. See Table III for a summary of the complexity of the various algorithms.

The main drawback of the mechanism is that it requires approximately three times as much storage as simple mechanisms. Fortunately, designers tend to focus their attention on a small portion of a layout at any given time; since corner stitching uses only local information, it will have good paging behavior in a demand-paged environment.

## APPENDIX A ADJACENCIES

The running time for several of the algorithms depends on the number of neighbors an individual tile has. One can construct situations where a tile has an arbitrarily large number of neighbors, so it is not possible to state any absolute upper bounds. However, graph theory can be used to determine the average number of adjacencies. In any connected planar graph

$$n - e + f = 1$$

where  $n$  is the number of nodes,  $e$  is the number of edges, and  $f$  is the number of faces contained by the edges. A face corresponds to a tile, a node to a corner of a tile, and an edge to a distinct adjacency between two tiles. For  $T$  tiles,  $f = T$ . The number of distinct nodes  $n$  can be at most  $4T$ , but in the interior of the tile structure, each corner of one tile must coincide with at least one corner of another tile (a "T" structure). Thus  $n \leq 2T$  and the total number of adjacencies is

$$e = n + f - 1 \leq 3T - 1.$$

Note that at the outside of the structure there may be corners that don't coincide with other corners, but for each of these there is also at least one edge that doesn't represent an adjacency (because there is no tile on the other side). Hence the  $3T - 1$  upper limit is not affected.

The  $3T - 1$  limit counts each adjacency only once for the two tiles that are adjacent. To compute the number of neighbors per tile, the figure must be doubled. This means that on the average, an individual tile will have about six neighbors, or about one or two on each side. This is regardless of the arrangement of tiles. Of course, if there are many tiles of different sizes, the large tiles may have many more than six neighbors. The average number of neighbors of a tile in a situation like this will be roughly proportional to the perimeter of the tile, which is less than linear in its area.

## APPENDIX B SPLITTING AND MERGING

This section discusses the cost of splitting one tile into two adjacent tiles, or merging two adjacent tiles into a single tile. A tile can be split into two tiles as follows:

- 1) Make an exact copy of the original tile.
- 2) Update the coordinates of each tile to reflect the split, and set the tiles' corner stitches to refer to each other.
- 3) Update the corner stitches in tiles that are now adjacent to the new tile. To do this, use the neighbor-finding algorithm to locate the neighbors on three sides of the original tile, then update the stitches that must point to the new tile.

The algorithm for merging two adjacent tiles into a single larger tile is similar: stitches must be updated along three sides of the tile that is eliminated.

The cost of each algorithm consists of constant factors (copying a tile or changing an  $x$  or  $y$  coordinate) and the search of neighbors on three sides. Appendix A showed that the number of neighbors was constant when averaged across a whole design, but increases for those tiles that are much larger

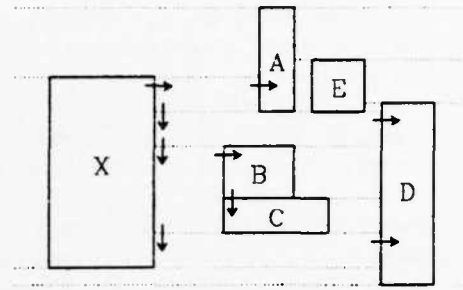


Fig. 18. An example of visibility searching. From tile  $X$ , tiles  $A$ ,  $B$ ,  $C$ , and  $D$  are visible to the right. Tile  $E$  is not visible from  $X$ . Tile  $D$  has two distinct windows of visibility to  $X$ , one between  $A$  and  $B$  and one below  $C$ . During a horizontal visibility search from  $X$ , the pictured corner stitches will be traversed.

than their neighbors. In this case, the average number of neighbors will be approximately proportional to the perimeter of the tile. Thus the cost of a split or merge is constant if the tile being split or merged is about the same size as its neighbors. If the tile is much larger than its neighbors, then the cost increases in proportion to the tile's perimeter, which is less than linear in its area.

## APPENDIX C VISIBILITY SEARCHING

This section gives algorithms that locate all solid tiles *visible* on one side of a given solid tile. Two solid tiles are mutually visible if it is possible to draw a horizontal or vertical line between them without crossing any other solid tiles. Fig. 18 gives examples of visible and invisible tiles. Visibility searching is used during compaction and stretching. Unfortunately, the horizontal-strip representation of space requires different algorithms for horizontal and vertical searches.

Horizontal-visibility searching is based on the neighbor-finding algorithm of Section V-5.2. The following algorithm is for searching on the right side of the original tile; it can be modified to search on the left side.

- 1) Use the neighbor-finding algorithm to enumerate the tiles that touch the right side of the starting tile. For each tile found, execute step 2) or step 3), depending on the tile's type.
- 2) If the neighbor is solid, then it is automatically visible.
- 3) The neighbor is a space tile. If it extends all the way to the edge of the circuit (infinity) ignore it. Otherwise, use the neighbor-finding algorithm once again to enumerate all the tiles that touch its right side. Each of these must be a solid tile. All of the tiles whose bottom edges are lower than the top edge of the starting tile are visible.

In this algorithm, a single solid tile may be enumerated several times, once for each distinct window of visibility with the starting tile (see, for example, tile  $D$  in Fig. 18). The time required for the horizontal search is linear in the total number of tile adjacencies in the search area, which was shown in Appendix A to be linear in the number of tiles. Since there must be at least one solid tile enumerated for every space tile enumerated, the expected running time of the search is linear in the number of solid tiles found. For tiles in a relatively uniform distribution, the number of visible neighbors



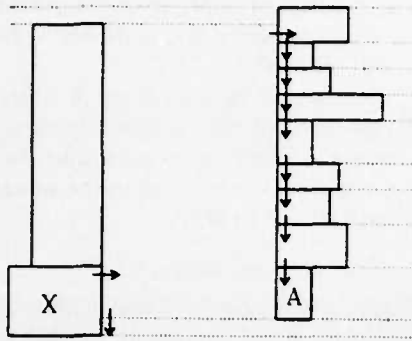


Fig. 19. A pathological case for horizontal visibility searching. When searching for tiles visible to the right of  $X$ , all of the tiles above  $A$  will have to be passed through and skipped over.

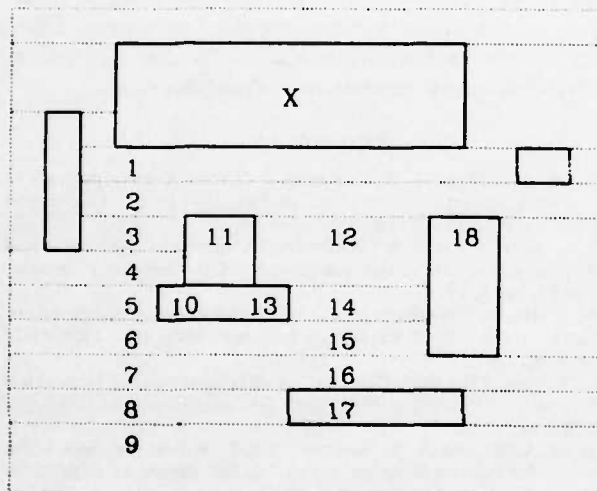


Fig. 20. In a downward visibility search from  $X$ , columns of space tiles are traversed until solid tiles are found or the end of the circuit is reached. In this case, the numbers give the order in which the tiles will be traversed (the numbering ignores realignments that must occur when advancing down the side of a column). Tiles that fall under more than one column are traversed more than one time.

of a given tile is small and independent of the size of the circuit. However, if a space tile found in step 3) extends above the starting tile, as in Fig. 19, it could have any number of out-of-range solid tiles along its right edge; since these have to be skipped over, the upper limit on the running time for the algorithm is the total number of solid tiles in the circuit.

For vertical visibility searches, the algorithm is an extension of the area-search algorithm of Section V-5.3. It consists of a recursive set of searches of successively thinner columns. The following algorithm will find all the solid tiles visible below the starting tile; it can be modified to find all those above the starting tile. See Fig. 20 for an example.

1) The initial column being searched extends downward from the bottom of the starting tile. Use the approach of the area-searching algorithm to advance one by one through the tiles lying under the left edge of this column.

2) When a space tile is found in step 1), check to see if it extends across the whole column. If so, then advance downwards to the next tile (this is the case for tiles 1 and 2 in Fig.

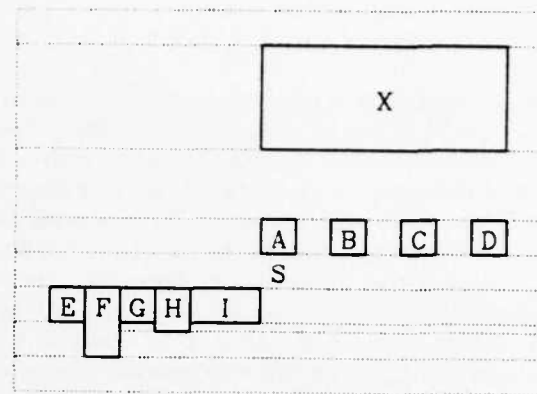


Fig. 21. Tile  $S$  causes severe misalignment during downward visibility searches from  $X$ . Each of tiles  $E-I$  will have to be traversed for each column between tiles  $A-D$ .

20). If the space tile extends downward to  $-\infty$ , then return.

3) If a solid tile is found, or if the space tile does not extend across the entire column, then do not continue down any further. Instead, scan across the top of the column (following its stitches from the tile found in step 1). Each of the solid tiles found in this way is visible to the starting tile. For each space tile found in this scan, invoke a recursive search on the column underneath this space tile (tiles 3 and 12 in Fig. 20 are examples of space tiles that start new column searches).

The algorithm terminates when all of the columns have been closed off by continuous solid tiles across the columns or when the end of the circuit (infinity) is reached. As with the horizontal search, tiles are enumerated once for each window of visibility with the starting tile. Since each of the visible tiles is visited once for each window of visibility, the expected running time is linear in the number of visible tiles (for relatively uniform tile distributions). However, the same misalignment that was illustrated in Fig. 9 for area searching can occur here, as shown in Fig. 21. In the unlikely event that most of the tiles in the circuit are piled up like tiles  $E-I$  in Fig. 21, they will all have to be traversed as part of each column, and the total running time will be proportional to the product of total circuit size and number of visible tiles.

Each of the visibility algorithms works in a particular direction. The directed nature is important to other algorithms that use visibility searches. The right visibility search enumerates visible tiles in order from top down, and the bottom visibility search enumerates visible tiles in order from left to right.

#### APPENDIX D PLOWING IN LINEAR TIME

The poor worst-case behavior of the plowing algorithm in Section 5.7 occurred because the algorithm processed tiles in a haphazard order. As a result, some tiles could be moved many times as the algorithm discovered that more and more space was needed to move other tiles out of the plow area. The linear-time algorithm makes two passes over the circuit. In the first pass, it computes how far each tile must move; tiles are processed in topological order so that a given tile

is not processed until its final position is known. In the second pass, the tiles are actually moved. Each tile is moved exactly once.

The linear-time algorithm requires an extra data value to be stored in each tile. This additional value is called the tile's *delta*, and gives the distance that the tile must be moved. Initially, all of the deltas are zero; when the plowing algorithm is finished, it leaves all the deltas zero for future plowing. The linear-time algorithm also requires the use of a linked list of tiles to be moved. Tiles are added to the linked list in the first pass; in the second pass, tiles are moved in list order.

Pass 1 consists of setting the delta of the initial tile to its plow distance and calling the following recursive procedure to process the tile. The basic algorithm is independent of the plow direction.

1) Add a pointer to the current tile onto the front of the list of tiles to be moved. This tile will be moved *before* all previously encountered tiles.

2) Use the tile's delta and location to compute the area that this tile will plow out as it moves.

3) Use the visibility search from Appendix C to enumerate all visible solid tiles in the plow area. Execute steps 4) and 5) for each neighbor tile found in this way.

4) Compute the delta required to move the tile out of the plow area. If this delta is greater than the tile's current delta, then update the tile's current delta.

5) If this is the last time we will ever see the neighbor, then call this procedure recursively to process the neighbor. The determination of "last time" depends on the directed nature of the algorithms for visibility searching. For example, in a left-to-right compaction, the "last time" is when the neighbor's bottom edge is in the window of visibility, or when the bottom edge of the overall plow area is in the window of visibility. The bottom edge of the overall plow area must be passed down in the recursive calls; it is the lowest bottom edge for any plow on the recursive stack.

Pass 2 scans the list in order from front to back. Each tile on the list is erased, then recreated at a new position determined by its delta. The ordering of the list guarantees that the final position of each tile is empty at the time it is moved. When moving the tiles, the deltas are zeroed out again in preparation for the next plowing operation.

If the total number of tiles moved is  $M$  and the total number of tiles in the circuit is  $N$ , then each of the two passes has an expected running time that is of order  $M$ , with worst-case running time proportional to  $MN$ . In pass 1, the recursive procedure is invoked exactly  $M$  times (once for each tile that must be moved). The overall running time for pass 1 is determined by the time spent in enumerating all the visible neighbors for all the tiles that are moved. In the average case, each tile's visible neighbors can be found in constant time, so the total

running time is proportional to  $M$ . In the worst case, the cost of the visibility searches may be  $MN$ , so the worst-case running time of pass 1 is of order  $MN$ .

For pass 2, the expected time to delete or create each tile is constant, so the expected running time is linear in  $M$ . However, the worst-case deletion or creation time for a tile is proportional to the overall circuit size, so the worst-case running time for pass 2 is of order  $MN$ .

#### ACKNOWLEDGMENT

Michael Arnold, Carlo Séquin, David Ungar, and David Wallace all took part in the discussions that led to the formulation of corner stitching. C. Séquin developed the proof that  $3N + 1$  space tiles are always sufficient in a design with  $N$  solid tiles. Gordon Hamachi, Bob Mayo, Walter Scott, and George Taylor implemented the layout editor based on corner stitching. In addition to these people, Leo Guibas, David Patterson, Alberto Sangiovanni-Vincentelli, and the referees all provided helpful comments on drafts of this paper.

#### REFERENCES

- [1] M. H. Arnold and J. K. Ousterhout, "Lyra: A new approach to geometric layout rule checking," in *Proc. 19th Design Automation Conf.*, pp. 530-536, 1982.
- [2] J. L. Bentley and J. H. Friedman, "A survey of algorithms and data structures for range searching," *ACM Computing Surveys*, vol. 11, no. 4, 1979.
- [3] M. Y. Hsueh, "Symbolic layout and compaction of integrated circuits," University of California, Berkeley, Tech. Rep. UCB/ERL/M79/80, Dec. 1979.
- [4] G. Kedem, "The quad-CIF tree: A data structure for hierarchical on-line algorithms," in *Proc. 19th Design Automation Conf.*, pp. 352-357, 1982.
- [5] K. H. Keller and A. R. Newton, "KIC2: A low cost, interactive editor for integrated circuit design," in *Dig. Papers for COMPCON Spring 1982*, pp. 305-306.
- [6] J. K. Ousterhout, "Caesar: An interactive editor for VLSI," *VLSI Design*, vol. II, no. 4, pp. 34-38, fourth quarter 1981.
- [7] J. K. Ousterhout, and D. M. Ungar, "Measurements of a VLSI design," in *Proc. 19th Design Automation Conf.*, pp. 903-908, 1982.
- [8] A. Sangiovanni-Vincentelli, private communication.



John K. Ousterhout received the B.S. degree in physics from Yale College, New Haven, CT, in 1975 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980.

Since 1980 he has been an Assistant Professor of Electrical Engineering and Computer Sciences at the Berkeley campus of the University of California. His research interests include computer-aided design, VLSI architecture, and operating systems.

PUBLICATIONS ON CIRCUIT & SYSTEM DESIGN



## CIRCUIT & SYSTEM DESIGN

The following section contains papers and reports relating to research in computer Circuit & System Design. They describe work which was wholly or in part performed under the sponsorship of the DARPA grant.

- (1) R. Kavalier, T. Noll, H. Murviet, M. Lowy and R.W. Brodersen, "A Dynamic Time Warp IC for a 1000 Word Recognition System," *Proc. of ICASSP*, San Diego, March, 1984.
- (2) P. Ruetz, S.P. Pope, B. Solberg and R.W. Brodersen, "Computer Generation of Digital Filter Banks," *ISSCC Digest of Technical Papers* Feb 1984.
- (3) S.P. Pope, B. Solberg and R.W. Brodersen, "A Single-Chip LPC Vocoder" *Tech. Digest of the ISSCC*, Feb. 1984.
- (4) C.C. Hsiao and R.W. Brodersen, "A Multirate Root LPC Synthesizer," *Proceeding of ICASSP*, March, 1984.

# A DYNAMIC TIME WARP IC FOR A ONE THOUSAND WORD RECOGNITION SYSTEM

Robert Kavalier, R.W. Brodersen  
University of California, Berkeley CA 94720

Tobias G. Noll  
Siemens AG, Munich Germany

Menachem Lowy  
GE, Schenectady NY

Hy Murveit  
SRI International, Menlo Park CA 94025

## ABSTRACT

Dynamic time warping is considered a superior way to perform time alignment in speech recognition. [1] Unfortunately dynamic programming algorithms require too much computation for conventional computer architectures to handle and still provide good response time with 1000 reference words. This paper presents a single chip that is capable of performing the dynamic time warp processing necessary for recognizing 1000 words in real-time.

## INTRODUCTION

MOS-LSI technology has made it possible to design circuits capable of processing a large number of complex operations on a single chip. This technology, when applied to the speech recognition task, allows one to design a chip capable of performing all the computations necessary to recognize words from a dictionary of 1000 words in real time using a dynamic time warp algorithm. In this case "real time" means that there is a very short (less than 20ms) delay from the detected end of the spoken word to the time that the recognition decision is made. The algorithm that was implemented is general enough so that connected speech can be recognized without any speed penalty. Also, a character recognition project uses this exact same chip for pattern matching.

Our chip does not use a general purpose architecture. Instead we used a very parallel and pipelined architecture and thus can run much faster than other chips that perform similar functions.

Before going any further we should define two terms used in this paper: template and frame. A template is a pattern that represents a typical way that a word or a phrase might be spoken. Templates are ordered sequences of frames, where each frame represents

short-term spectral information.

## THE TIME WARP ALGORITHM

Time alignment using a dynamic programming algorithm is a very popular means of providing high accuracy in many current speech recognition systems. Unfortunately, these algorithms run very slowly when implemented on a standard computer architecture. [2, 3, 4] If no constraints are placed on the system then each frame of an incoming unknown utterance must be compared to each frame of each template in the dictionary. For each frame-to-frame comparison, one must compute a Euclidean distance between two N-element vectors, find the minimum among 3 numbers, and add that minimum to the Euclidean distance. This process takes many instructions on most computers, so pruning techniques were used to eliminate the need to compute all possible frame-to-frame comparisons.

Our chip (the time warp chip) does not prune paths as a method of eliminating frame-to-frame comparisons. Instead, we have built a chip that has the power to compute all of these comparisons very quickly using a pipelined and parallel architecture. The additional circuitry needed to perform the pruning would have been much too expensive. The comparisons are computed on a column by column basis thus allowing one to start processing an unknown utterance before it is spoken completely.

The algorithm implemented is similar to those presented by Sakoe and Chiba. [5] The distance between two words is defined as  $D(a_{1..M}, b_{1..N})$ :

$$d_{j,k} = \sum_{i=0}^{j-1} (a_i - b_k)^2$$

$$D_{j,k} = \min(D_{j-1,k}, D_{j,k-1}, D_{j-1,k-1}) + d_{j,k}$$

The boundary values of  $D$  are fixed to be infinite along the template word axis and programmable along the unknown word axis. This allows a connected-word algorithm to be implemented without additional hardware. [6] No pruning

### TEMPLATE MEMORY TIME-WARPING PATHS

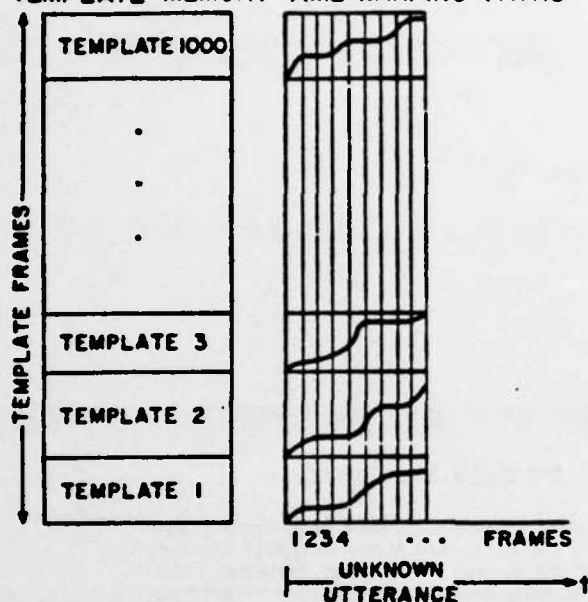


Figure 1: Computation

or global slope constraints are used, thus each frame of the unknown word must be compared to each frame in the template word memory. Local slope constraints can be applied as a programmable option.

### SYSTEM ARCHITECTURE

The time warp chip is the processing element in a more complex system. An entire speech recognition system is being designed on a single Multi-bus board. The system consists of the time warp chip, a digital filter bank chip,<sup>[7]</sup> a general-purpose processor (used for higher level processing, or as a general-purpose computer), and enough memory to store over 1000 templates. The time warp chip interfaces to the general purpose processor through a dual-ported memory, a dma-driven parallel port, and a general parallel port. The dual ported memory contains the templates, the boundary value for  $D$  (bottom score), and the unknown word frame. The bottom score and unknown word frame are updated before each column is computed. The dma-driven port allows the general purpose processor to catch the final scores between each template and the unknown word. The other port tells the chip to start a new column.

For 1000 words one needs enough memory for 25000 frames (25 frames per word). We have set aside enough space for 32000 frames (using 32 standard HM4864P-2 64Kx1 dynamic RAMs). In addition to the template memory a scratch-pad memory is needed for the

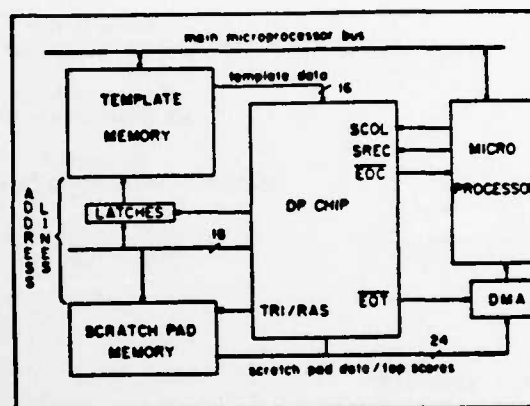


Figure 2: System Block Diagram

dynamic programming calculations. This memory is made with 12 16Kx4 chips (TMS4416-15). The general-purpose processor has its own memory (128K bytes), allowing it operate independently of the time warp chip.

### CHIP OPERATION

The time warp chip has 68 pins:

- 3 - Power, Ground, Substrate
- 16 - Input data from template memory
- 24 - I/O data for scratch pad memory
- 18 - Address lines (shared) for template and scratch pad memories
- 3 - Control line inputs:
  - Start First Column (SCOL)
  - Start Other Columns (SREC)
  - Clock
- 4 - Control line outputs:
  - End of Column (EOC)
  - End of Template (EOT)
  - RAS for scratch pad Address
  - Read/Write from scratch pad memory

The chip starts up in an idle mode, where it performs sequential reads from template and scratch pad memories to refresh dynamic memories. When either a SCOL or SREC signal is received the address counter goes to 0 and reads the first word of template memory into the bottom score register. The next 3 words are ignored. Then one frame (4 words) of the unknown utterance is read into an internal memory. Next the chip performs the time warp algorithm with the remaining templates. The end of a template is indicated by the special code FFFF hex as its first word. If the second word is FFFF then the end of column has been reached. At the end of column and the end of template the appropriate outputs are strobed to allow scores to be retrieved by the general purpose processor. At the end of a column the chip returns to the idle mode.

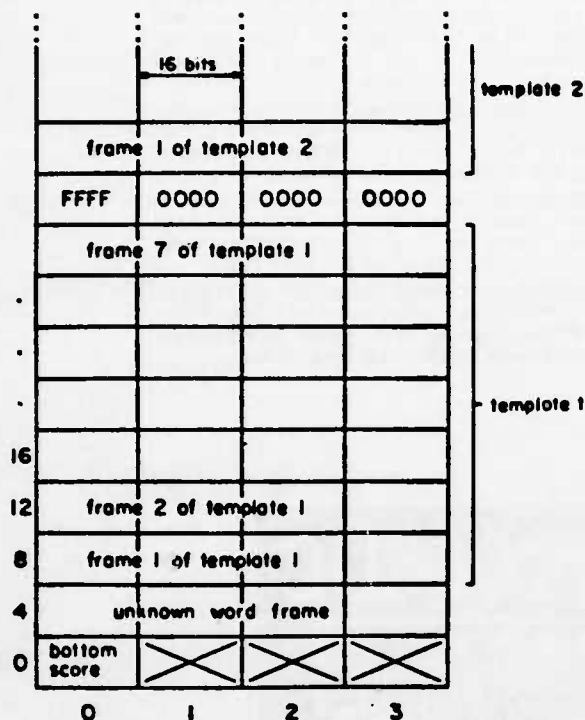


Figure 3: Memory Organization

### CHIP ARCHITECTURE

The chip consists of 5 functional units:

- 1) A distance processor that can compute a 4-dimensional Euclidean distance every clock cycle.
- 2) A pipeline accumulator that sums four 4-dimensional Euclidean distances into one 16 dimensional distance.
- 3) A dynamic programming processor that can compute one minimization and sum every 4 clock cycles.
- 4) An addressing unit for the external template and scratch-pad memories.
- 5) A controller for each of the above processors.

Each processor has a custom designed architecture. The controller is a combination of custom designs and standard PLA finite state machines.

The distance processor has a four level pipeline. First four 4-bit differences and absolute values are computed in parallel. Second these differences are squared (using a PLA) resulting in four 8-bit values. Third, these 8-bit values are summed pairwise into two 9-bit values. Finally a 10-bit sum is computed. This value is saturated to 8-bits. The dynamic programming processor also

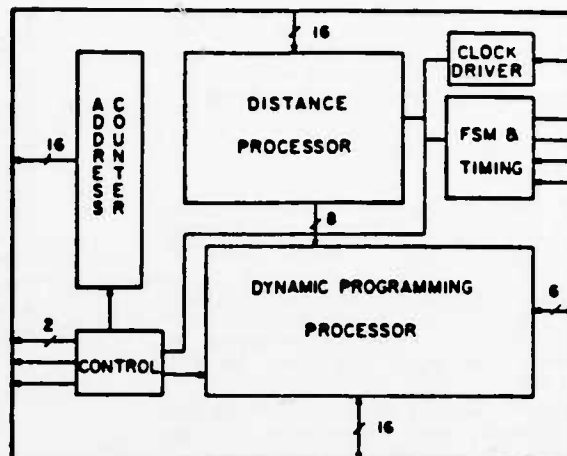


Figure 4: Chip Block Diagram

computes the projection of the path on the unknown utterance axis. This is needed in connected speech applications. The projection is computed with an 8-bit saturating counter.

Four of these differences are then accumulated into an 8-bit register. The resulting 8-bit sum is then sent to the dynamic programming processor.

The dynamic programming processor has three 16-bit registers corresponding to all possible paths into a given element of the DP matrix. One of the 16 bits is used for slope constraints. Two of these registers are fed from memory, and one is an accumulator. To process a node (frame) one must first compute the minimum of these three registers, then sum in with the distance above. Three comparators and a PLA are used to compute the minimum of the registers. The PLA contains the rules for handling the first row and first column of the DP matrix. The summing output is saturated to 15 bits to prevent overflow.

Due to bandwidth considerations, the address counter must count up 2 then down 1. The new DP sums are written after the decrement, the old DP sums are read after the increment. This sequence requires a special counter.

The system is controlled with a standard finite state machine. There is also a circuit that computes various pipeline timing signals, and a high speed counter for internal synchronization.

The chip is implemented in a 4 micron NMOS process, has an active area of 20,000 square mils, and runs with a 5MHz clock.

### CONCLUSIONS

An NMOS LSI chip was designed that has enough processing power to perform the dynamic programming algorithm used to recognize 1000 words in real time. This chip can be used in either isolated word or connected word applications. The speed of chip was accomplished by using a highly parallel and pipelined custom design.

#### REFERENCES

1. L. R. Rabiner and S. E. Levinson, "Isolated and Connected Word Recognition - Theory and Selected Applications," *IEEE Trans. on Communications* Vol. COM-29 pp. 821-859 (May 1981).
2. D. J. Burr, Bryan Ackland, and Neil Weste, "A High Speed Array Computer for Dynamic Time Warping," *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 471-473 (Spring 1981).
3. *Texas Instruments Digital Signal Processor, TMS320 Product Description*, Texas Instruments P.O. Box 1087, Richardson Texas, 75080 (1983).
4. H. Ishizuka, M. Watari, H. Sakoe, S. Chiba, T. Iwata, T. Matsuki, and Y. Kawakami, "A Microprocessor for Speech Recognition," *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 503-506 (Spring 1983).
5. Hiroaki Sakoe and Seibi Chiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Trans. Acoustics, Speech and Signal Processing* Vol. ASSP-26 pp. 43-49 (Feb. 1978).
6. John S. Bridle, Micheal D. Brown, and Richard M. Chamberlain, "An Algorithm for Connected Word Recognition," *Proc. Int. Conf. Acoustics, Speech and Signal Processing* Vol. 2 pp. 899-902 (May 1982).
7. Peter Reutz, Steven P. Pope, Bjorn Solberg, and R.W. Brodersen, "Computer Generation of Digital Filter Banks," *Proceedings of the IEEE International Solid-State Circuits Conference*, (Feb. 1984).

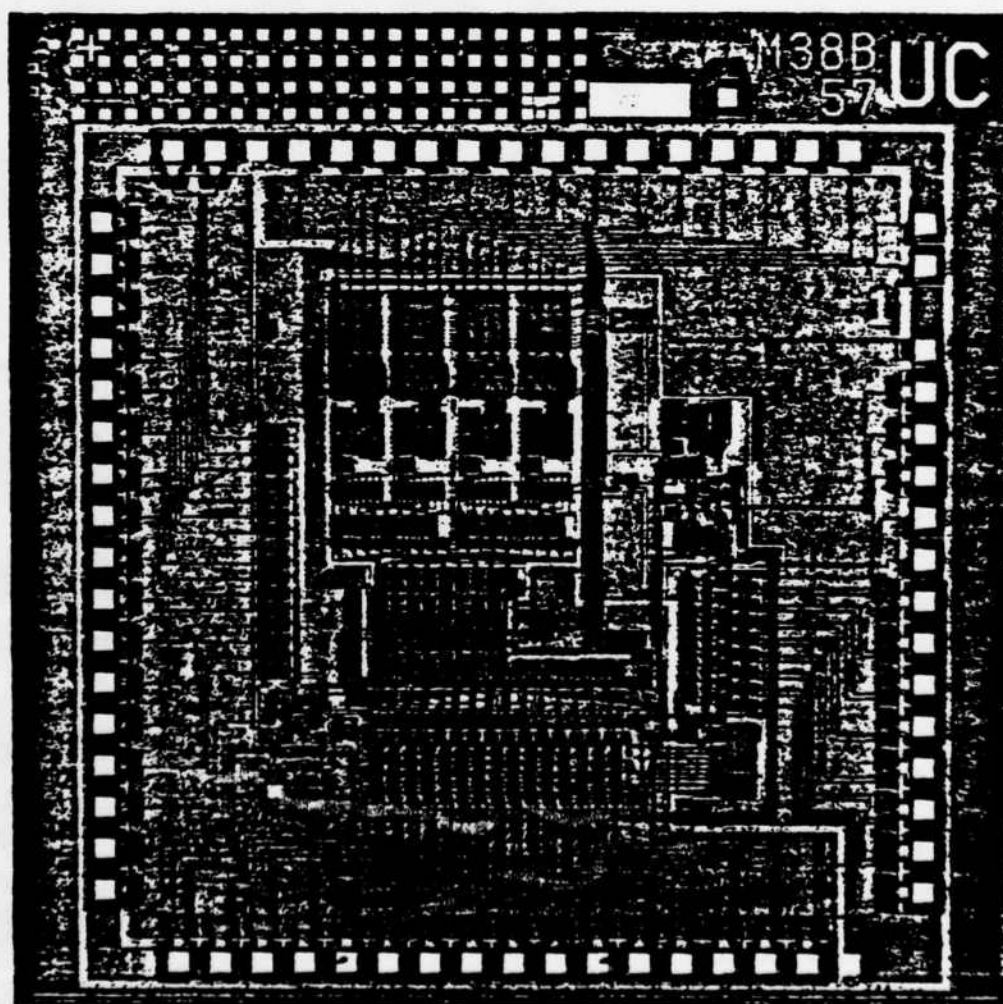


Figure 5: A micro-photograph of the dynamic programming chip

## Computer Generation of Digital Filter Banks

*Peter A. Ruatz, Stephen P. Pope, Robert W. Brodersen*

University of California, Berkeley  
Electronics Research Lab  
Berkeley, California 94720

### ABSTRACT

In order to reduce the design time of digital filter bank circuits, a design system has been developed. The software consists of the filter compiler which converts high level filter descriptions to hardware descriptions and the layout generator which converts the hardware descriptions to a layout file. To verify the algorithms before fabrication, a test system is employed. The development time of this system was kept to a minimum by designing the hardware to be easily micro coded and assembled. Several circuits have been fabricated and tested that were generated with this system, including a single band pass filter chip, a 112 pole 16 channel filter bank for a speech recognition system and a 16 channel spectrum analyzer for consumer stereo applications. The speech recognition chip achieved a SNR of 80 dB with an area of 25 sq mm in a 4 micron NMOS technology.

March 30, 1984



## Computer Generation of Digital Filter Banks

*Peter A. Ruetz, Stephen P. Pope, Robert W. Brodersen*

University of California, Berkeley

Electronics Research Lab

Berkeley, California 94720

### 1 INTRODUCTION

Fully automated design of complex integrated circuits has often resulted in limited usefulness because of poor performance or inefficient silicon space utilization. If few restrictions are placed on the function of the ICs to be generated, then the optimization problem becomes difficult, yielding circuits far inferior to custom designs. Another important aspect of using automated design systems is the time required to develop the software and its reliability. There is no advantage in reducing hardware design time if the resultant software development effort becomes equally time consuming and error prone.

It has become apparent that tradeoffs between development time, generality and final circuit performance must be made. The design system described here was based on an emphasis on high performance with minimal software effort. Instead of treating the software and hardware designs as distinct problems, the hardware architecture and layouts were designed in a way that made automation simpler while maintaining performance.

Some automated design systems have been developed which allow the user to interact only at the highest level. If this is incompatible with the requirements of the user, then the entire design system is of no use. If however, the software is designed to allow the user to operate at a lower level, more jobs can



be accomplished. Our design system has been developed in a hierarchical manner. For those wishing to generate filter banks, the task can be accomplished from the highest level, i.e. totally automated. The user can also use the lower levels of the system (i.e. only partially automated) for other applications. Further, the system can be extended at the highest level for the specific needs of the user.

The scope of applications that has been chosen is digital filter banks which are a parallel and/or cascade connection of filter sections. Digital filter banks are found in applications as diverse as MODEMs and spectrum analyzers for speech recognition, channel vocoders, consumer stereo and EEG analysis. Decimation and rectification are required in addition to digital filtering in the spectrum analysis applications.

## II THE HIERARCHY

Currently, the hierarchy is four levels deep. At the lowest level are the circuit 'cells'. These cells consist of basic building blocks such as counters, adders, RAM cells, ROM cells, etc. The cells can be used without any automation for a totally manual design. At the next level, the layout generator assembles these cells into more complex blocks such as data paths and controllers from hardware descriptions. This would be useful for users that desire a signal processor, but need a few additional circuits that have not been designed or are not handled by the layout generator. The user would only have to specify the hardware specifications including the RAM size and ROM contents and add the new circuit blocks to form a completed chip. At the third level, the layout generator assembles the data path and the controller into a completed chip. For those that have a non-filter digital signal processing application, a chip could be generated completely from this hardware description. Finally, at the highest level, the filter compiler generates the hardware description from a digital filter

- 3 -

description. At this level, digital filter banks can be generated completely automatically.

To generate filter bank chips, the design procedure shown in figure 1 is followed. The digital filter bank structure and coefficients are specified in a input file. The filter compiler converts the input file to the hardware description. To check the algorithms before the circuit is fabricated, the hardware description can be used as an input to a real-time tester. When the designer is satisfied, the layout generator is used to create a layout file.

### III THE CELLS

The basic architecture of the hardware is shown in figure 2. There are two main blocks: the controller consisting of the program counter, the ROM and the address index register and the data path consisting of the ALU and RAM.

There are several reasons for having few large circuit blocks. The block division was chosen to minimize assembly difficulty while retaining adequate generality. With few blocks, the automatic assembly is simplified. Routing difficulty is reduced by having fewer blocks that need to be routed together. The blocks are also made up primarily of abutting circuit cells which are very simple to assemble.

The large blocks were chosen to be functionally complete. That is, the blocks can be easily used to perform some complex function. The blocks would be complicated to use, except at the lowest level of cells, in a partially assembled form. The program counter may be useful without the ROM but it is easily assembled from counter cells so that it need not be a separate block.

There is also a somewhat natural division. As all cells in the data path could be designed with the same bit slice pitch, the data path could be made a single block requiring no data bus routing. The ROM was designed to minimize area

that determined the pitch of ROM cells. The ROM cell pitch is, however, vastly different from the pitch of the control lines entering the data path. That makes it more efficient to optimize each as separate blocks with routing between the two than to stretch the ROM to the pitch of data path control lines.

### **III-1 The Controller**

The controller was designed to be small with high performance. To achieve these goals it was made very simple with a minimum number of features. For example, there is no branching capability or micro coded instructions. Adding complexity can result in vastly increased area as the extra registers and routing are a significant fraction of the controller. ROM bits are very small, regularly spaced and hence very efficient. Instead of putting the convenience of micro coded instructions in hardware, it is put in the software (at the highest level) where it does not add to the silicon area.

Every cycle the controller outputs a valid horizontal control word. This horizontal control word specifies the value of every data path control line. Each controller output comes directly from the ROM with the exception of some of the RAM address lines when decimation is used. With decimation, the index register modifies the RAM address. Although this increases controller complexity, it saves ROM space and averts the need to perform address computations in the data path. The data path is never used for any control operation, allowing continuous signal processing. The circuit is also more compact since busses are not needed to connect the control and data path.

### **III-2 The Data Path**

Figure 3 is a block diagram of the data path. As in most signal processors there is a RAM, adder, accumulator, some form of negation/absolute value logic and i/o. However, no array multiplier is included.

Again, only a minimum of features are provided. In this way the size can be kept small making room for additional data paths on a single chip for greater throughput. The cell circuit design problem is also reduced, while programming the data path is more complicated. This is not a problem when automation is used, as the filter compiler generates and optimizes the micro code.

Since there is no array multiplier, fixed coefficient multiplies are implemented in a serial-parallel manner [1]. This is accomplished with the use of the barrel shifter, adder and accumulator. Since a restriction of fixed coefficient multiplies is placed on the system, less than  $N$  cycles are required for an  $M \times N$  multiply, where  $M$  is the signal width and  $N$  is the multiply coefficient length, by programming the ROM properly. Because a barrel shifter can shift several (0 to 5 in this case) places in a single cycle, multiplies require only a number of cycles equal to the number of '1's in the coefficient.

By using coefficients encoded in canonical signed digit format [2], it is possible to save more cycles in a serial-parallel multiply when there are more than two consecutive ones in the coefficient. This arises because in hardware it is just as easy to subtract as it is to add a partial product. For example:

to perform:  $(0.01111)Y_n$

rewrite:  $=((0.1)-(0.00001))Y_n$   
 $= (0.1000(-1))Y_n$   
 $= (0.1)Y_n - (0.000001)Y_n$

Therefore, only two cycles are necessary to perform the multiplication, whereas four cycles would be required for the direct implementation.

The adder is a ripple-carry type. To ensure high speed, different even and odd cells are used which minimizes the delay through the carry chain. This results in very compact circuits which can operate at rates over 4 MHz. The

ripple carry adder is also particularly well suited for a bit slice design which makes the automatic layout very straight forward. The output of the adder saturates instead of simply overflowing to prevent limit cycles. This is easily incorporated in the hardware but would require several cycles per computation to implement in software.

Pipelining in the data path increases the performance of the circuit by making higher clock rates feasible. The pipeline registers are at the output of the RAM, the input of the RAM and at output of the adder (the accumulator). With pipelining, the RAM and the barrel shifter, adder combination both get a full cycle for operation. Although pipelining makes micro coding more difficult, it is transparent to the user when the filter compiler is used.

The memory input register is the only register of the three which can be selectively written. In some cases, the result of a computation can be held in this register until the RAM is inactive during a serial-parallel multiply. At this point, the result can be written into the RAM without requiring an extra cycle. Proper use of this register reduces the length of the micro code by preventing the data path from becoming memory bound.

The RAM is a four transistor dynamic type with a schematic shown in figure 4. A dynamic memory was chosen over static designs because the dynamic RAM is smaller with lower power consumption. The RAM is automatically refreshed as long as the sample rate is kept over 1 KHz because every location is both written and read each sample.

Three possible choices for the RAM design were the one, three or four transistor cells. The four transistor cell was chosen over one transistor designs to minimize process sensitivity. To avoid running busses between the RAM and ALU, it was desired to have the same pitch for both so they could be attached directly, simplifying automatic layout. To use space efficiently, this required

that the RAM have a single column decode as the optimized cell pitch was approximately half that of the ALU bit slice. The three transistor design is more difficult to column decode so the four transistor design was chosen.

#### **IV LAYOUT GENERATOR**

The layout generator assembles the cells into a data path and a controller block from hardware descriptions. If desired these blocks are then assembled into a complete chip. The hardware is described by several parameters including: data path word width, RAM size, decimation ratio and ROM contents.

##### **IV-1 Layout Generation Issues**

Before starting development of the layout generator, several aspects of automated layout were identified as difficult problems. General placement of the major circuit blocks requires sophisticated optimization algorithms to generate space efficient designs. A two level router would be needed to route between these blocks. An extensive data base would be needed to store the necessary data for the router and placer. The data base would contain the terminal locations on each block and the available routing area.

Other aspects of the automated layout were found to be easily handled problems. It is not difficult to assemble blocks (ie the ROM, ALU and RAM) from abutting cells since the relationships involved are all well determined by the hardware parameters specified by the user and the cell characteristics. The way the cells go together is determined by the cell designer so that proper cell design can help the automated layout. For example, by including the signal routing within the cells, the need for inter-block routing by the program is avoided. Fixed routing, where the routing terminals have a constant relationship throughout all changes in hardware parameters, can be accomplished by inserting a cell with the appropriate wires in it. That is, no algorithm for routing

is required at all. Regular routing, where the routing terminals are evenly spaced throughout changes in the hardware parameters, is implemented by a simple program loop.

#### **IV-2 The Floor Plan**

In order to avoid the more difficult problems, two major restrictions were made. The first was to use a fixed floor plan, the relative placement of circuit blocks, pads and routing areas on the chip. The floor plan was chosen to reduce the complexity of the algorithms used and the number of layout decisions that must be made by the program. With the chosen floor plan all routing is either fixed or regular.

The decision was also made to have the program 'know all'. That is, all information regarding the cells and their connection was coded directly into the algorithms. Using specific information of the application avoids having to solve the general problem and reduces the software design time. Software reliability is enhanced when the simplest algorithms are used instead of complex general algorithms with obscure failure modes. This approach obviously makes the programs very specific to the particular cells which are used so that changes in the cells may require changes in the software. Therefore, one should not expect to make major upgrades without significant software changes with a system such as this. However, because the software development time is relatively small, new software can be written when significant changes are made.

#### **IV-3 Examples of Generated Circuits**

The circuit remains easy to assemble over the large changes in hardware parameters shown in figure 5a-d. The hardware parameters for each is listed in table 1. From the figure the fixed floor plan can be seen. The controller, data path, pads and routing areas are always in the same relative position. The I/O



parallel buss at the top of the chip is an example of regular routing. The routing area does not change shape or relative position as the parameters change. The routing between the controller and the data path is a function only of the RAM size and whether decimation is used. As there are only a few cases, each is treated as fixed routing and a cell with the appropriate wires is simply inserted. Wiring from the PC to the ROM is handled similarly. The wiring of supplies and clocks requires little jumping (except in the fixed routing cells) and a minimal amount of decision making.

The silicon area is also used efficiently over the range of parameter changes. Virtually the only wasted area is near the pads or due to differences in length of the controller and data path (see figures 5b,5c). The RAM gets longer as the number of states in the filter bank increases. The controller increases in length with the program length. Since adding states requires a longer program to process these states, the ROM and RAM tend to get larger together. In figures 5a-c the ROM is not column decoded and the waste area is not too large. In figure 5d the ROM length increased significantly so that a column decoded ROM was used to minimize the unused space.

Some waste of space is allowed if the waste is not large while the savings in effort is. For example, when decimation is used, the ROM width is constant regardless of the RAM size. Up to 3 bits of RAM are unused but the routing is simplified. The data buss routing area between the data path and pads on the right side of the chip is of constant size. These simplifications reduce the number of cases to be handled with some space wasted for the very small chips. However, the designs would likely not be used anyway, due to the large overhead involved.

#### IV-4 Output File Format

The output of the layout generator are KIC format [3] files. This format was chosen because layout stations are being used which read this format making visual checks convenient. The KIC format also supports the hierarchical organization of the hardware. The CIF format [4] is used for actual fabrication but does not allow arrays as the KIC format.

#### IV-5 Block Assembly

As mentioned previously, the assembly of blocks from cells is a straightforward task. An output file is written that lists the cells with the appropriate offsets and orientation. This information is calculated from the hardware parameters and cell parameters (eg size).

The controller is a connection of many cells that makes its manual layout difficult. Most variations in the controller are functions of the ROM width and length (found from the binary listing), and the decimation ratio all of which the user specifies explicitly. The ROM length determines how many bits will be used in the PC and decoder and how the decoder is programmed. Since there are only 5 different PC sizes, each is a cell with appropriate routing wires. The decimation ratio determines which type of ROM output register will be used and how the index register itself is configured. If there is no decimation, all output registers are the same and no index register is used. If there is decimation the output registers that feed the index register input are of a different type and an index register must be included and programmed to decimate properly.

The data path assembly is quite simple due to the bit sliced nature and the small number of blocks. The entire ALU only requires one line in a KIC file specifying an array of bit slices. The entire RAM array is similarly specified. The RAM decoder can be generated in the same way as the ROM decoder with each cell

being described by one line in the KIC file.

## V THE FILTER COMPILER

The filter compiler generates hardware descriptions from digital filter descriptions. This allows the automatic generation of filter banks with virtually no knowledge of the final hardware.

### V-1 Filter Specification

The compiler reads an input file specifying the filter bank organization in terms of a parallel connection of channels. Each channel is a cascade connection of sections. Variations on this format are allowed that have been found useful in some applications. A section can be factored out and used by different channels. An example of this is the direct form band pass filter. The zeros are the same for all channels and can be factored out and computed only once. Figure 6 shows an example of a filter bank organization. In this example there are 16 parallel channels, each consisting of a 4th order BPF section, rectifier, 1 pole LPF section, decimation by 8 and a 2nd order LPF section.

Each section is a single input, single output structure with delays, multiplications and additions. Diagrams of some of the currently programmed sections are shown in figure 7.

All multiplies defined in the sections use fixed coefficients of canonical signed digit format. Use of this format, which was described earlier, optimizes the usage of the adder by minimizing the number of cycles required to perform multiplication.

There are several options allowed in each section in the bank. The user can full wave rectify the input of any section. This is useful in spectrum analyzer applications. Decimation is also handled but in a somewhat restricted way. A number of channels can have their outputs decimated and modified by some

specified filter. The post decimation filter is the same for all channels being decimated and the decimation ratio is always the same as the number of channels being decimated. These restrictions were applied simply to reduce the development time and could be relaxed in future systems. The user can also specify that the output of any section be sent off chip through the parallel buss while setting an output strobe. To implement multiple inputs, the input of any section can be taken from any channel output or any channel input. Being able to specify a channel output, allows the use of a filter by many other channels (described above) and really allows very arbitrary filter organizations. Normally, the default (no specification) results in the parallel channels operating on the same input data.

The format of the input file is tailored to filter banks and was chosen to simplify the compiler. The format is as follows:

1. Input channels (one or more)  
These sections receive data from off chip and may perform some filtering (eg zeros of direct BPF).
2. Standard channels (one or more)  
These are just the regular channels, ie some cascade connection of sections. These channels will be decimated if a decimation channel is specified.
3. Decimation channel (optional)  
This is the channel that operates on the output of all standard sections above after decimation.
4. Non-decimated Standard channels (optional)  
More regular channels that are not to be decimated.

The format for the the sections is as follows:

1. Section identifier (2 letters), <options, if any>, N coefficients

The format specifies the order that micro code is generated and stored in the ROM and hence the order that it is executed. This save the compiler from having to determine this information.

## V-2 The Filter Library

The compiler references a filter library which contains pertinent information about the allowed sections. A file contains a list of valid section identifiers along with the number of memory locations and coefficients required for each section.

For each section there is also a file containing the macro for that section. The macro file contains the symbolic micro code that implements a section without the coefficients or options inserted. Symbolic micro code is just a description of data path control lines that have been grouped functionally. The symbolic micro code has fields to describe the following:

- memory operation
- relative memory address (actual address computed by compiler)
- barrel shifter input mux selection
- number of shifts (constant or taken from input file)
- adder a input mux select
- adder b input mux select
- i/o operation

Currently, this micro code must be written by hand for each section. This involves a detailed knowledge of the timing and architecture that the average user would not have. Although software could generate the micro code from difference equations, this was not chosen because higher performance code could be generated by hand. For a second order section the length of the micro code is typically only 8 words.

## V-3 Compiler Operation

The operation of the filter compiler is shown in figure B. On the first pass, the input file is checked for errors and hardware requirements such as RAM size and decimation ratio are determined. On the second pass, symbolic micro code for the entire bank is generated. The symbolic micro code is then compressed. Finally, the symbolic micro code is assembled to binary micro code.

During the first pass, several errors are checked for, the amount of RAM is determined and each state is assigned a RAM location. The error check will locate syntax errors, undefined sections, filter library errors or the use of the wrong number of coefficients. If decimation is used, the decimation ratio is determined by counting the number of standard sections before the decimation section. The RAM requirements can then be determined. Without decimation, the amount of RAM required can be found by simply adding up the memory requirements for each section. With decimation, things are not as simple. The index register supplies the high order RAM address lines when the decimation is performed so that some RAM may not be used.

$$\begin{aligned} \text{Amount of RAM accessed } A = & (\text{RAM needed for input and standard sections}) \\ & + (\text{RAM needed for decimation channel}) \\ & * (\text{number of decimated channels}) \end{aligned}$$

$$\text{Amount of RAM included on chip } B = 2^{(\text{int}(\log_2(A-1))+1)}$$

Each state is then assigned to a RAM location. The states are assigned to sequential RAM locations as they are encountered in the input file if there is no decimation. That is, the first state of the first filter is stored in the first RAM location while the last state of the last section is stored in the last location. With decimation, the states accessed by the decimation filter are assigned first at fixed increments. The remaining states are then filled in sequentially.

The symbolic micro code for the entire bank is generated during the second pass. To accomplish this the input file is scanned until a section declaration is found. The macro for that section is read from the library and expanded into complete micro code by inserting the coefficients and options from the input file. This process is repeated until the end of the input file is found.

The symbolic code generated in the second pass is compressed by looking for sequences of code that can be shortened. There are three cases that are optimized. First, and most important, is the performing of the first memory

access during the final computation of the previous section. This appears at nearly every section boundary and utilizes the pipelining of the data path. Another case is the utilization of both adder inputs when the accumulator is empty. Normally the B input is zeroed and data is brought in through the A input. If a coefficient has certain properties, additional data can be brought in through the B input, saving one cycle. One cycle can be save if data needed for the next operation is found to be left in the adder during the previous calculation. This occurs for certain filter structures with some coefficients.

These optimizations help produce code with essentially the same efficiency as that done by hand. For a speech recognition filter bank, the number of micro instructions was reduced from 480 to 384 words. The optimization is performed on the symbolic code because the cases are easier to identify than when the code has been assembled to binary.

With the symbolic code optimized, it is converted to binary for use by the layout generator. This is a simple operation because for each symbolic field value there is exactly one binary pattern for one or more control lines. This data is written directly to a file for the layout generator or tester control data is included for use by a real time tester.

## VI THE TESTER

The tester set-up in shown in figure 9 is quite valuable in producing designs which work the first time fabricated. The filter compiler running on the VAX 11-780 generates tester code that is down loaded to a pattern generator. The pattern generator performs exactly the same function as the controller block included in the complete chip. It sends the horizontal control words in real time to a data path that is the same as that used in a final chip. The spectrum analyzer generates digital input data and examines the filter outputs. In this way, the filter designer can check the input file for errors and the effects of



finite data word with and coefficient truncation on the filter responses. If there is a problem, it is found before fabrication. Further, this set up will verify that the compiler is working properly and that the filter library data is correct.

## **VII FABRICATED CIRCUITS**

Several circuits have been fabricated using this system. A single band pass filter chip was fabricated to determine the efficiency of a small chip. A 16 channel filter bank for the front end of a speech recognition system and a 16 channel consumer stereo spectrum analyzer have been generated and fabricated. Table 2 gives a summary of the performance of these chips.

All circuits have been fabricated with a four micron NMOS depletion load process and are designed to work with a single 5 V supply. Although a 3 MHz non-overlapping clock is sufficient for the chips to operate at the designed sample rates, they can be run reliably with clocks up to 4 MHz.

### **VII-1 A Small Chip**

The 4 pole band pass filter chip die photo is shown in figure 10a and measured frequency response in figure 11a. Although the area per pole for this chip is quite high it might be useful when data is in digital form so that a switched capacitor or other analog filter would not be appropriate because of the high overhead in including the A/D and D/A. The circuit shown has a word width of 10 bits and a dynamic range of 48 dB. Since each additional bit increases the dynamic range 6 dB, a chip with 100 dB of dynamic range would only be 30 % larger.

### **VII-2 A Spectrum Analyzer for a Speech Recognition System**

A block diagram of 112 pole speech recognition system [5] chip is shown in figure 6. Each channel consists of a 4 pole Butterworth band pass filter, followed

by a full wave rectifier and the first pole of a 3 pole Butterworth low pass filter. The output of the 1 pole anti-aliasing filter is decimated and low pass filtered with the rest of the Butterworth filter. A photo of the die is shown in figure 10b. The frequency response of all 16 channels is shown in figure 11b.

The number of cycles available to perform all filtering is given by:

$$\text{number of micro instr} = (\text{number of processors}) * (\text{max clock rate}) / (\text{sample rate})$$

To ensure that this maximum number of operations was not exceeded, several steps were taken. Filter structures were carefully chosen. The state variable form shown in figure 7a has a relatively complex structure compared to the direct form (figure 7b). However, the state form is less sensitive to coefficient truncation than the direct form when there is a large ratio of sample frequency to filter band edge frequency. For low frequency filters, the insensitivity to coefficient truncation makes the state form filter more efficient than the direct form. At high frequencies, the direct form becomes more efficient due to its simpler structure. Therefore, the five lowest frequency filters are state form while the upper eleven are direct form. To save more cycles, the zeros of the direct form were factored out of each channel and computed only once. In the state form, at low frequency the zero at  $1/2$  the sample frequency has little effect and was left out.

### VII-3 A Spectrum Analyzer for Consumer Stereo

The structure of the 16 channel consumer stereo spectrum analyzer is very similar to that of the speech recognition chip. The sample rate was increased to 20 KHz to allow higher frequency filters and the band pass filters were limited to 2 poles. The  $1/2$  octave filters range in center frequency from 45 Hz to 8 KHz. The ratio of the lowest frequencies of interest to the sample rate is extremely small (much worse than the speech recognition chip) indicating that the state form will be better at lower frequencies. The photo of the die is shown in figure

10c with the log-log frequency response shown in figure 11c.

The design of this chip was automated one more level than the others. Instead of specifying the digital filters, a program was written to generate the digital filter specifications from desired 3 dB frequencies. The program picked the most suitable structure and determined and truncated all coefficients.

## VII CONCLUSIONS

The tools discussed here have been extremely valuable in the development of the circuits that have been fabricated. These tools not only shortened the hardware design time, but provided testing that found all errors before fabrication. Minor changes, such as increasing the width of the data path and fine tuning the gains of the channels, were made by simply editing the filter description file. Normally this would be a tedious task prone to careless mistakes. By careful design of the circuit cells and restricting the applications to filter banks, the software complexity was reduced with a development time of one man-month.

Table 1 Parameters for the Circuits of Figure 5.

	circuit 5a	circuit 5b	circuit 5c	circuit 5d
	1 channel	8 channel	16 channel	16 channel
RAM length (words)	8	64	64	64
data path word width	10	16	16	20
ROM length	32	128	128	192
Decimation ratio	-	8	8	8
number of processors	1	2	2	2

Table 2 Performance Summary for three Circuits.

	single 4 pole	16 channel speech recognition	16 channel consumer hi-fi
data path word width	10	20	20
size	2.8mm x 2.5 mm	7.2mm x 3.7mm	6.7mm x 3.6mm
power dissipation	260 mW	570 mW	570 mW
SNR	48 dB	80 dB	80 dB
number of poles	4	112	80
sample rate	84 KHz (max)	14 KHz	20 KHz

#### References

- [1] L. Rabiner, B. Gold, Theory and Applications of Digital Signal Processing, Prentice Hall, 1975.
- [2] L. Schmidt, "Designing programmable Digital Filters for LSI implementation", Hewlett Packard Journal, Vol 29, no 13, p. 15-23.
- [3] K. Keller, A. Newton, "KIC 2: A low-Cost, Interactive Editor for Integrated Circuit Design", Proc. 24th COMPCON, Feb 1982.
- [4] C. Mead, M. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.
- [5] M. Lowy, et al, "An Architecture for a Speech Recognition System", ISSCC DIGEST OF TECHNICAL PAPERS, p. 118-119, Feb 1983.
- [6] R. Agarwal, C. Burns, "New Recursive Filter Structures Having very low Sensitivity and Roundoff Noise", IEEE J. Circuits and Syst., vol CAS-22, pp. 921-927, Dec 1975.
- [7] P. Ruetz, et al, "Computer Generation of Digital Filter Banks", ISSCC DIGEST OF TECHNICAL PAPERS, p. 20-21, Feb 1984.

This research was sponsored by DARPA under contract number N00034-K-0251.

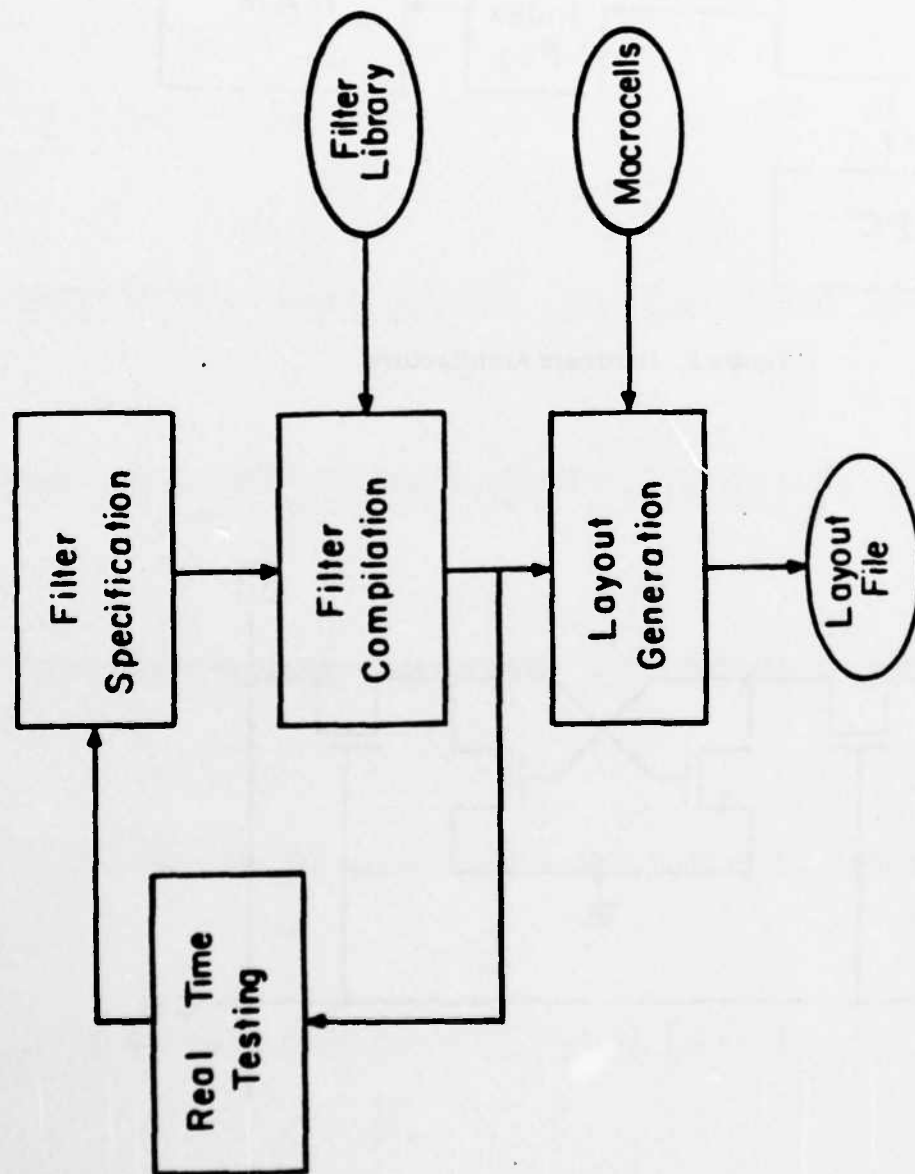


Figure 1. Filter Bank Design Procedure

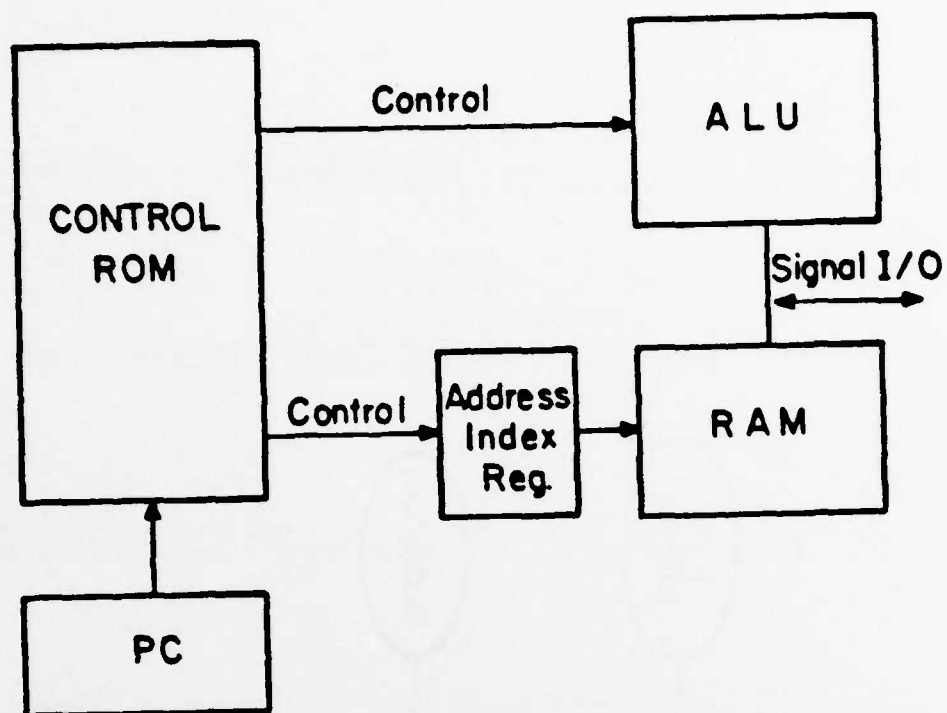


Figure 2. Hardware Architecture

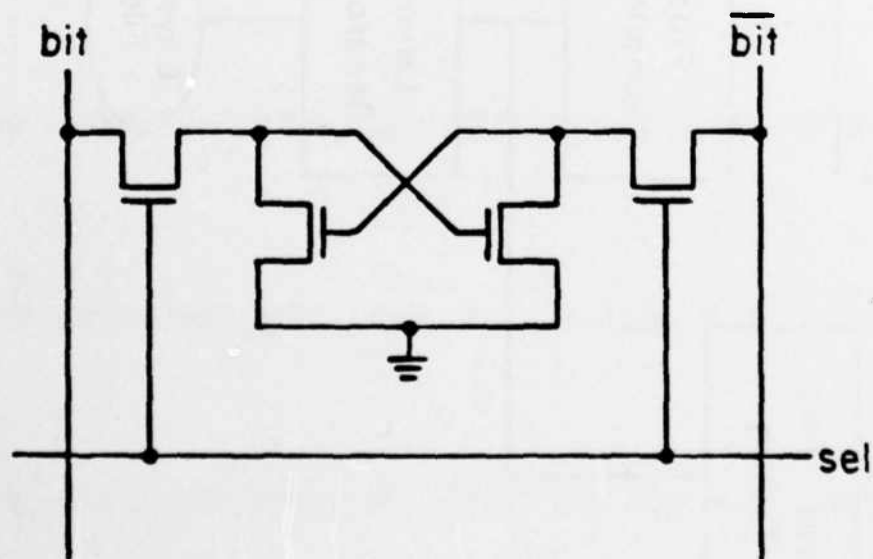


Figure 4. 4 transistor RAM cell



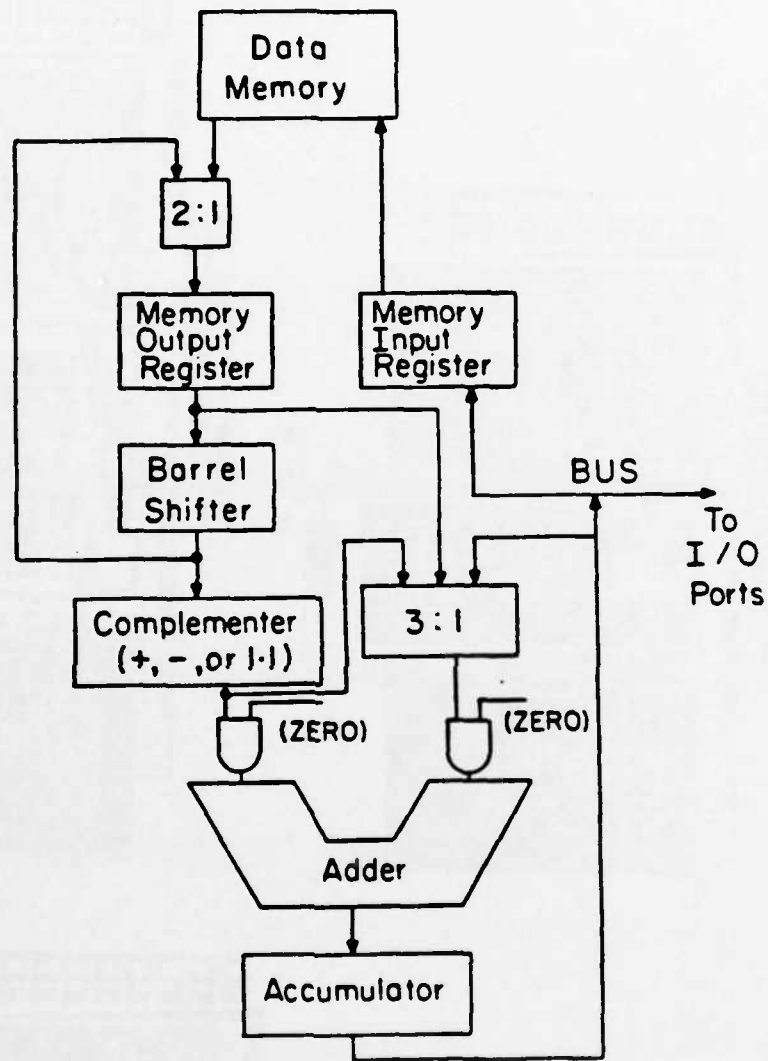


Figure 3. Data Path Block Diagram

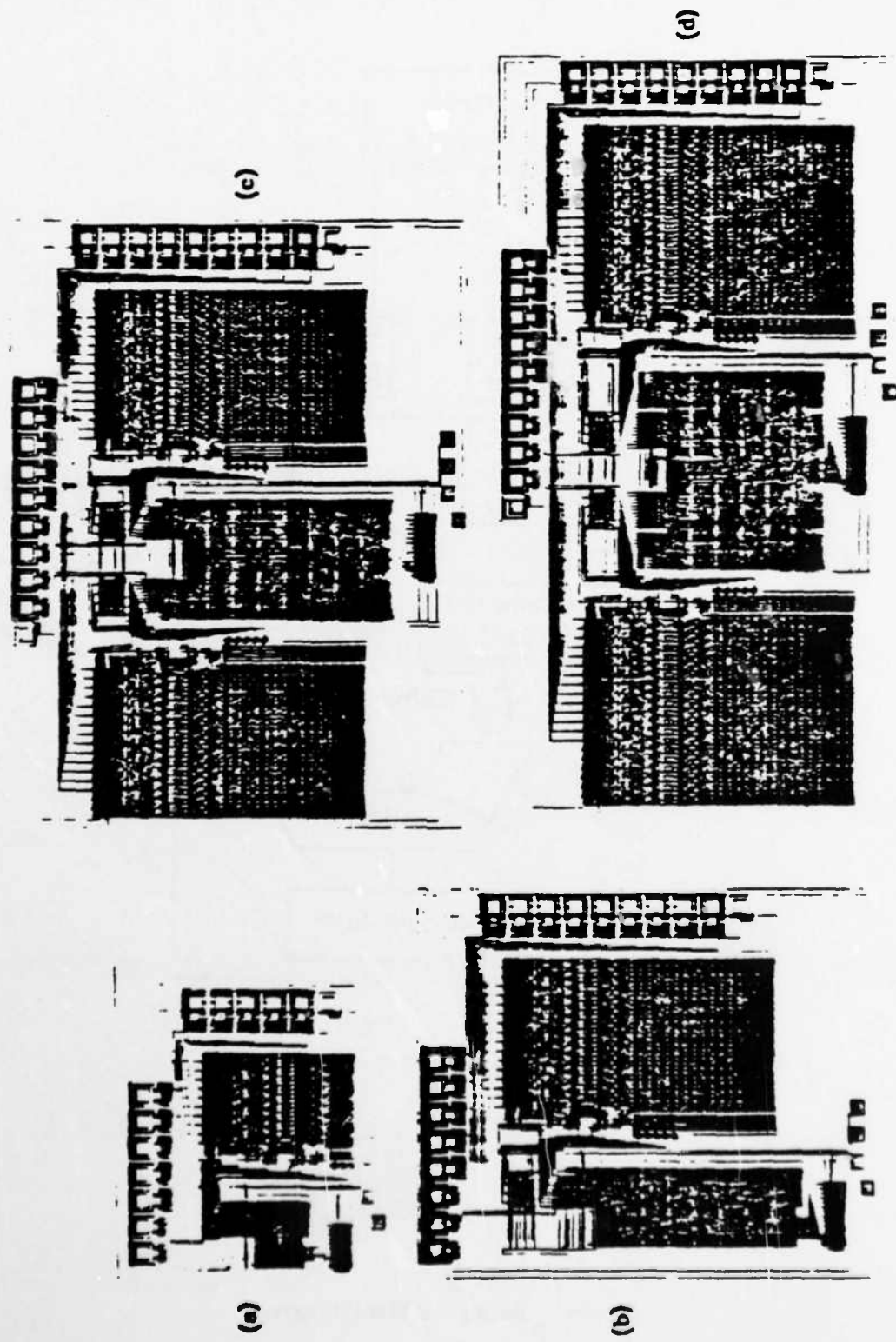


Figure 5. Layout Generator Output (see table 1)

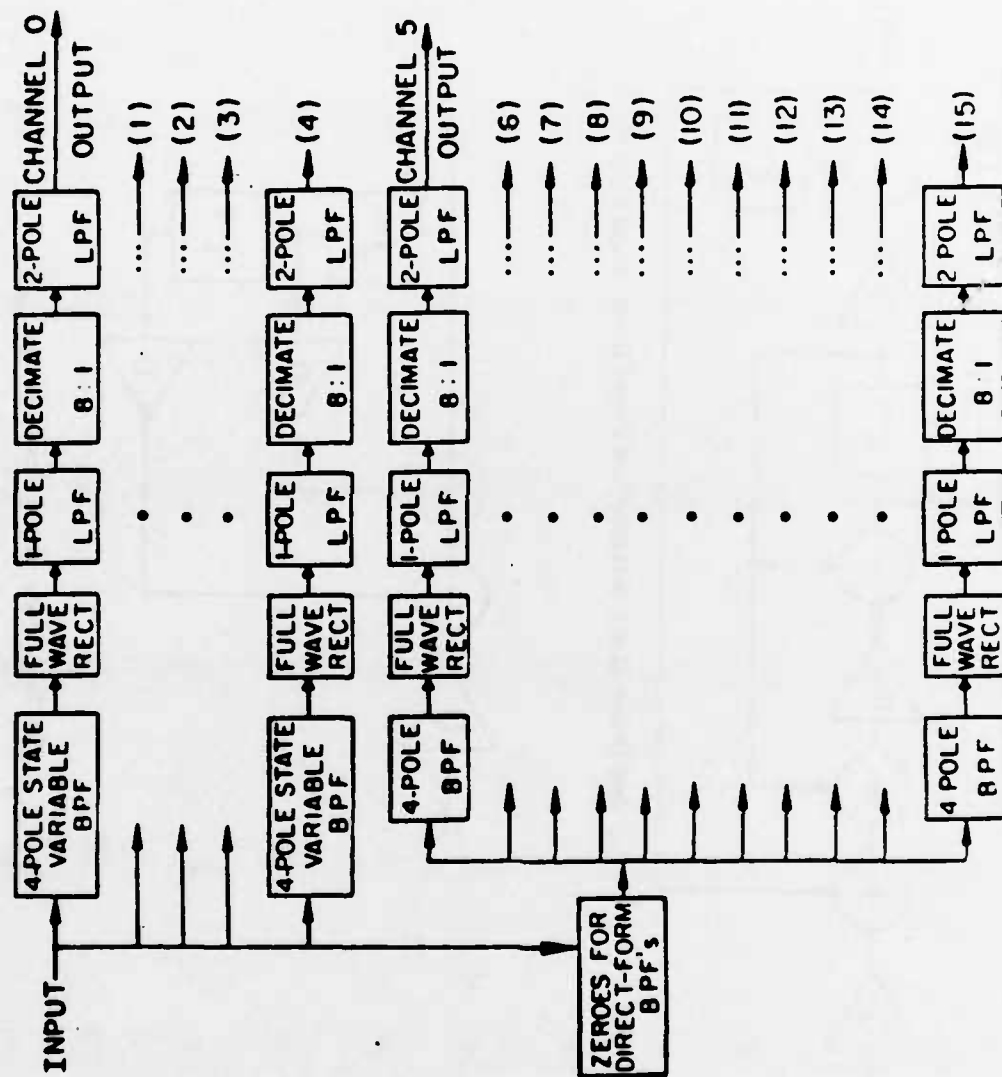
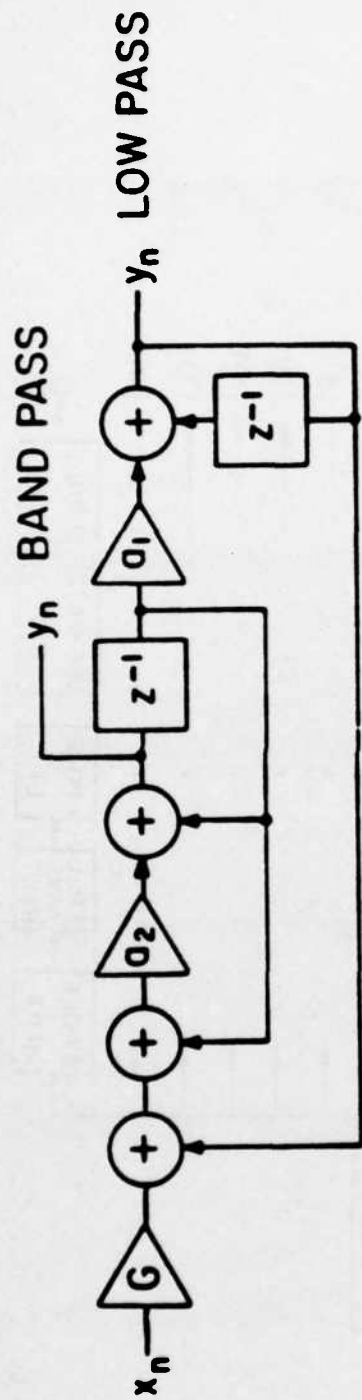
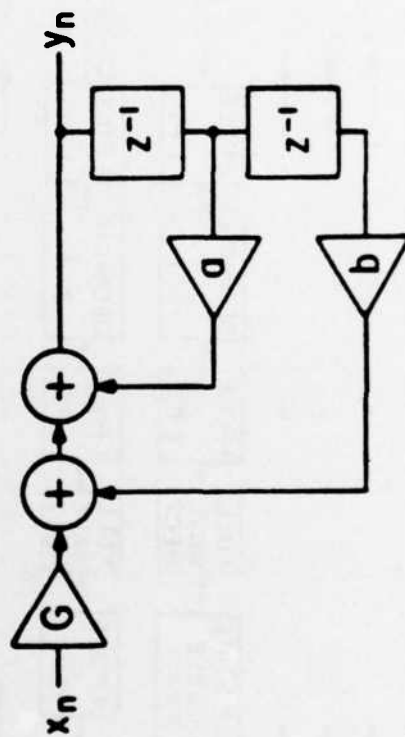


Figure 8. Speech Recognition System Filter Bank Organization

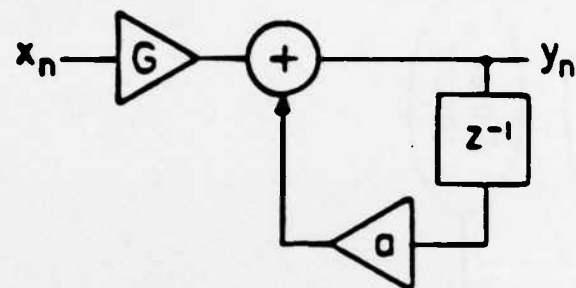


(a) 2nd Order State Variable Low Pass Filter, Band Pass Filter

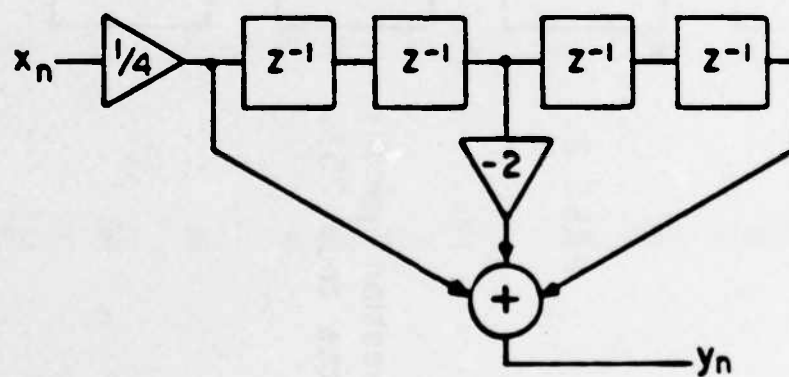


(b) 2nd Order Direct Form Poles

Figure 7. Filter Structures used in the Speech Recognition Chip



(c) Single Pole Low Pass Filter



(d) 4th Order Zeroes for Direct Form Band Pass Filters

Figure 7. Filter Structures used in the Speech Recognition Chip

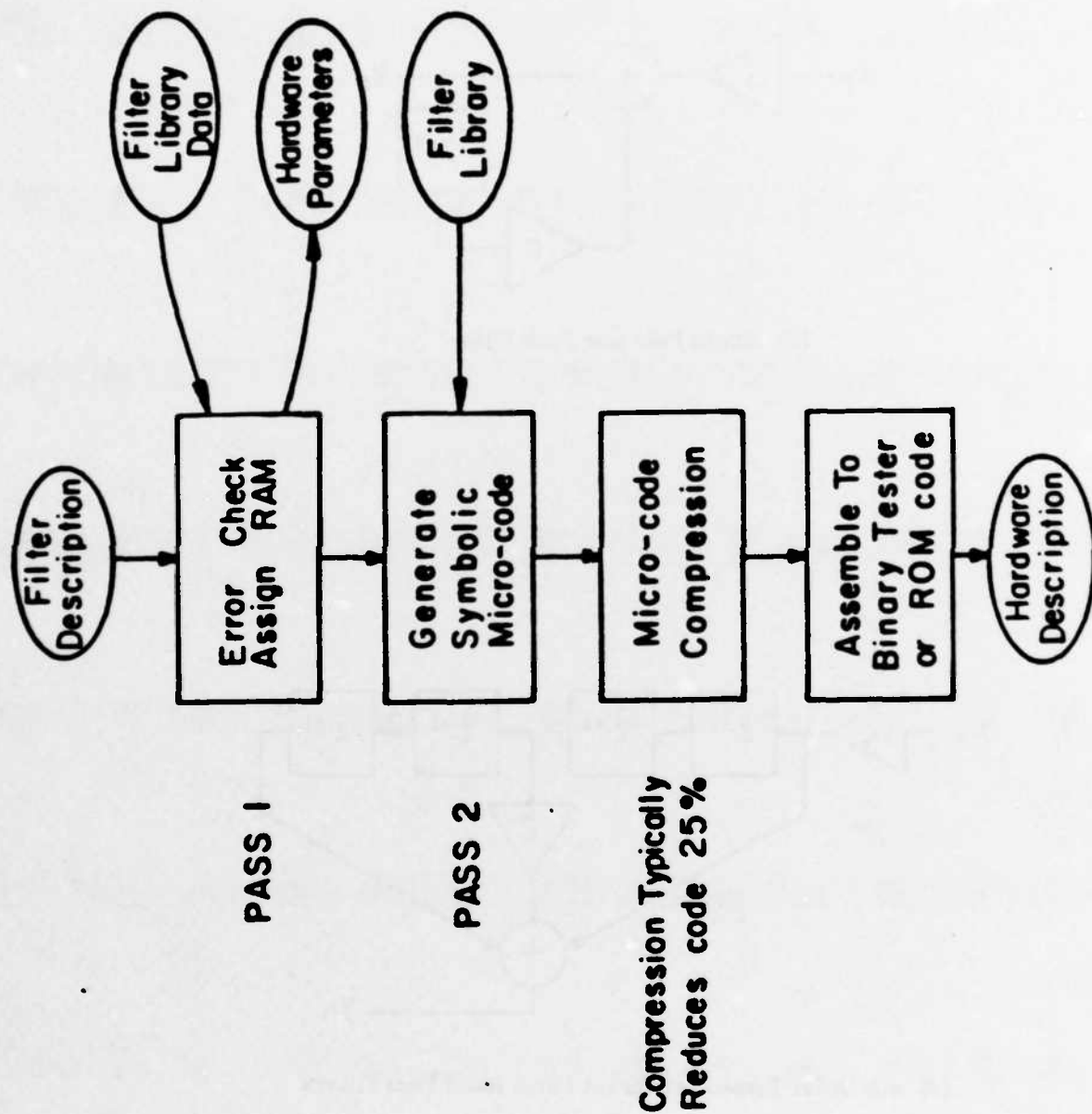


Figure 8. Filter Compiler Operation

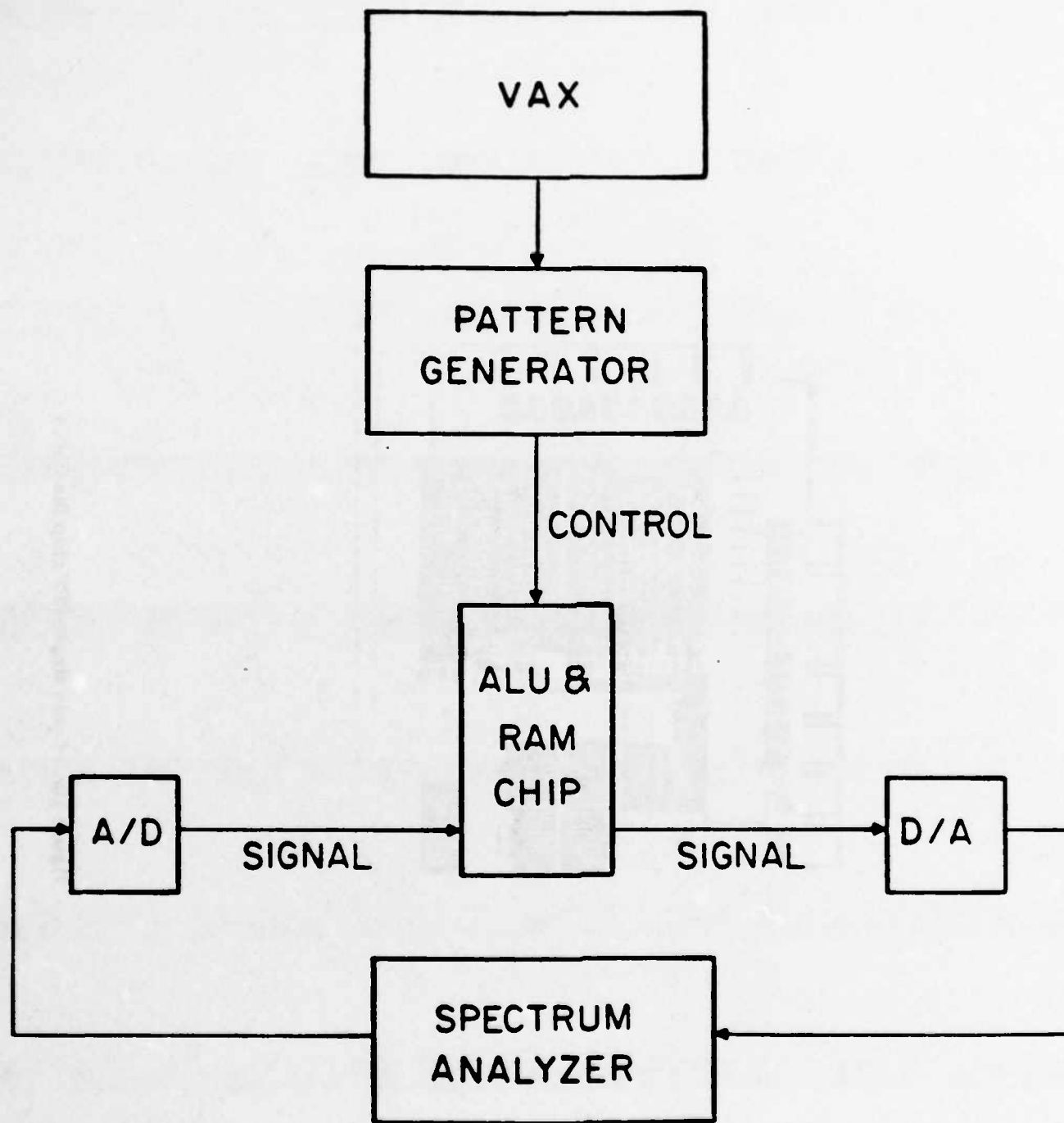


Figure 9. Test System for Verifying Algorithms



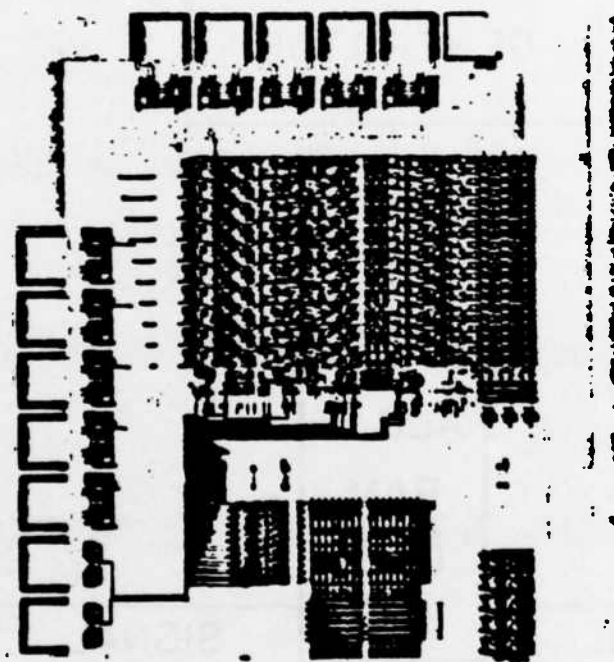


Figure 10a. 4 pole single HPF chip Die Photo

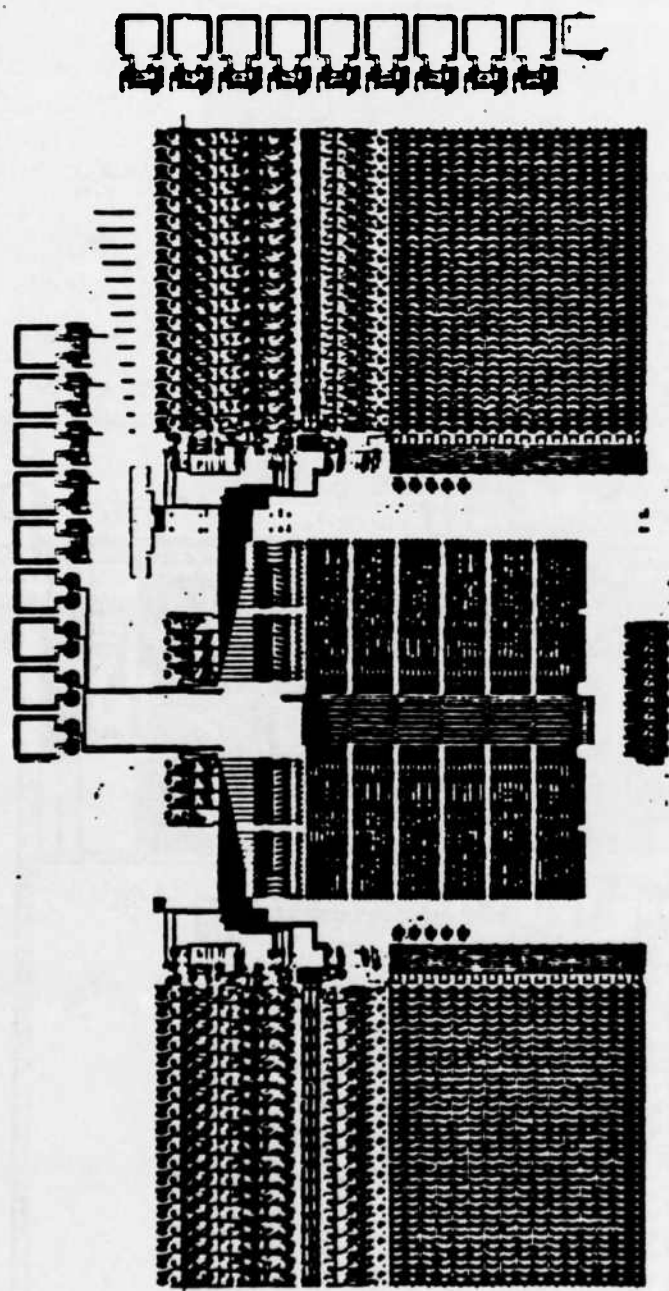


Figure 10b. Speech Recognition chip Die Photo

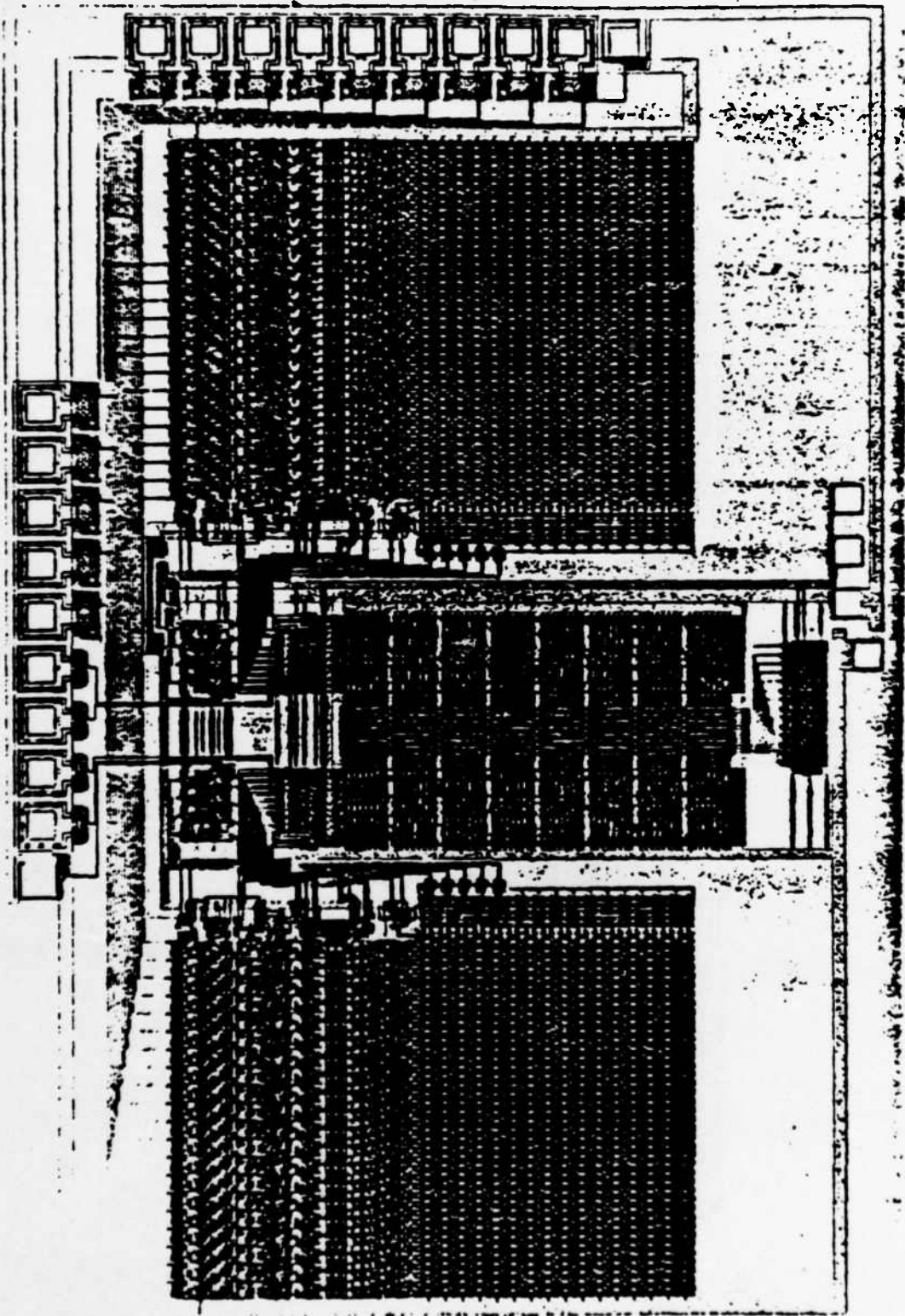


Figure 10c. Stereo Spectrum Analyzer chip Die Photo

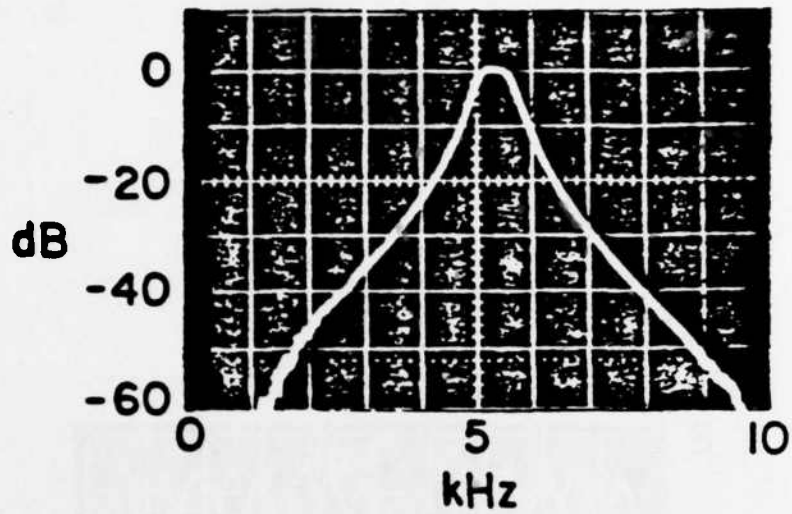


Figure 11a. 4 pole single BPF Frequency Response

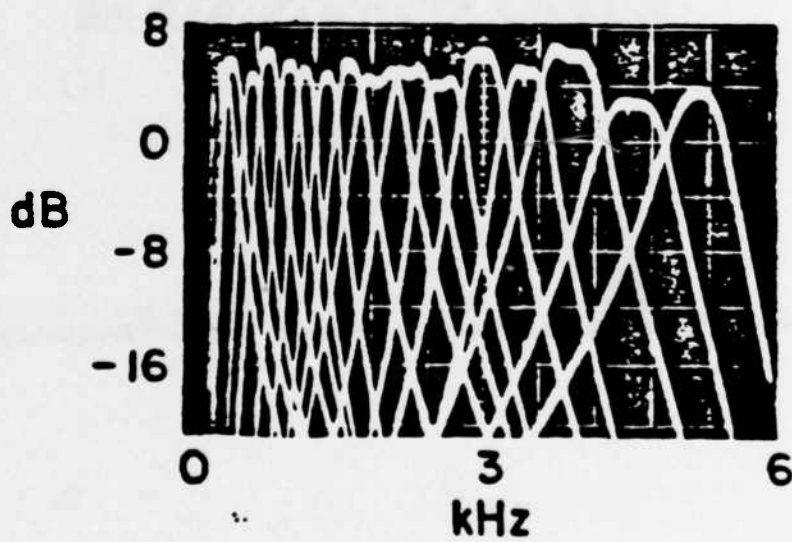
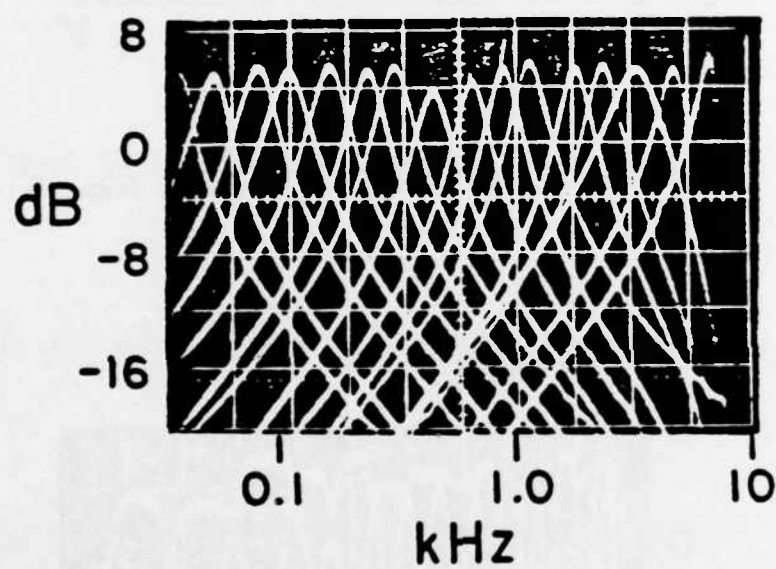


Figure 11b. 16 channel Speech Recognition chip Frequency Responses



**Figure 11c. 16 channel 1/2 octave spectrum analyzer Frequency Responses**

## A Single-Chip L.P.C. Vocoder<sup>\*</sup>

Stephen P. Pope  
Bjorn Solberg<sup>†</sup>  
Robert W. Brodersen

Department of Electrical Engineering and Computer Science  
University of California, Berkeley Ca. 94720. (415) 624-1779

A digital MOS-LSI circuit which implements a full-duplex speech analysis/synthesis system will be reported. This vocoder I.C. analyzes speech in real time, generating a low-bit-rate digital data stream suitable for transmission or storage. Simultaneously, synthesized speech can be generated from an incoming data stream.

Vocoders transmit two types of information: spectral parameters and excitation parameters. The vocoder I.C. uses linear predictive coding (L.P.C.) to represent the spectrum. The excitation is represented by its energy, a voiced/unvoiced decision, and the period of the pitch fundamental.

The vocoder I.C. contains three processors, each dedicated to a particular part of the algorithm (Fig.1). Most resources are devoted to the spectral analysis and pitch tracker. The synthesizer requires relatively little processing, using about half the resources of Processor No. 1.

Each processor is configured with a word length and data memory size appropriate to its function. The use of parallel processors, pipelining, and bit-serial communications all contribute to area efficiency.

Many methods exist for performing an L.P.C. analysis. Most group the input data into blocks (e.g. 128 samples) from which parameters are extracted. The alternative is the adaptive approach which updates the parameters every sample. Adaptive methods are more economical since there is no need to buffer a block of input. Also, the frame rate used for transmitting the parameters need not be tied to the block size in an adaptive approach.

In the vocoder I.C. the L.P.C. analysis is performed by a ten-stage adaptive lattice analyzer [1,2]. This requires both an all-zero filter (Fig.2) and a correlator (Fig.3). The reflection coefficients  $k_i$  are formed by taking the normalized cross-correlation of the signals  $A_{i-1}$  and  $B_{i-1}$ . This minimizes the energy of the outputs  $A_i$  of each stage. In the frequency domain, the filter response closely matches the inverse of the input spectrum.

The operation of the lattice analyzer is illustrated in Fig.4. The data was obtained by observing the data bus of Processor No. 1. As the signal passes through the ten stages of the lattice, its energy decreases and its spectrum flattens. The output of the final stage (the residual) is essentially white. All useful spectral information is represented by the ten reflection coefficients.

Although not used for low-bit-rate transmission, the availability of the residual permits use of the vocoder I.C. in higher bit-rate systems which encode the



residual.

Among the advantages of the lattice approach is the low coefficient sensitivity. Short (8-bit) word lengths may be used for the reflection coefficients, reducing the hardware required for multiplication.

The pitch tracker uses Gold's algorithm [3]. Peaks and valleys in the low-pass filtered speech are detected. The current input sample and the levels of the most recent peak and valley are combined in different ways to form six signals. These six signals are fed to six identical pitch detectors.

Each pitch detector attempts to time the interval between peaks in its input. Detection of a peak is followed by a blanking interval during which new peaks are ignored. Smaller peaks whose amplitudes fail to exceed an exponentially decaying threshold are also ignored. The threshold level is derived from the amplitude of the last peak detected. Counting the number of samples between peaks gives an estimate of the pitch period. The operation of one of the six pitch detectors is shown in Fig.5, obtained from Processor No. 3. The six pitch estimates thus obtained are combined with a "scoring" algorithm. A voiced/unvoiced decision is also made.

The complete vocoder algorithm is encoded in 6 kbit of microcode. A sequencer containing the microcode ROM controls the three processors. Certain operations which can not be efficiently microcoded are implemented by specially designed circuits. An example is the squaring operator used in the correlator (Fig.3), which is done by table look-up. The following circuit sections are identified by number in Fig.6:

- (1) Excitation source for synthesizer
- (2) Address indexing unit for Processor No.1
- (3) FIFO buffer for reflection coefficients
- (4) Squaring circuit
- (5) Address indexing unit for Processor No. 3
- (6) Decision-making logic for pitch tracker

The circuit is fabricated in a 4u NMOS process, containing 23,000 transistors on a .265" by .225" die. The circuit requires a 2.88 MHz clock and dissipates 600 mW.

† Research supported in part by Defense Advance Research Projects Agency, Contract No. MDA903-79-C-0429.

‡ Currently with the Norwegian Defence Research Establishment.

#### References:

1. Kang, G.S., "Application of Linear Predictive Encoding to a Narrowband Voice Digitizer", Naval Research Laboratory Report 7779, Washington, D.C. 1974.



2. Fellman, R.D., Hurst, P.J., Brodersen, R.W., "Switched-Capacitor Circuits for Adaptive Filtering and Autocorrelation", International Solid State Circuits Conference Digest, pp. 126-127, New York, 1993.

3. B. Gold, R.L. Rabiner, "Parallel Processing Techniques for Estimating Pitch Periods of Speech in the Time Domain", J. Acoust. Soc. Am., V. 46, No. 2, Pt. 2, pp. 442-449, August 1974.

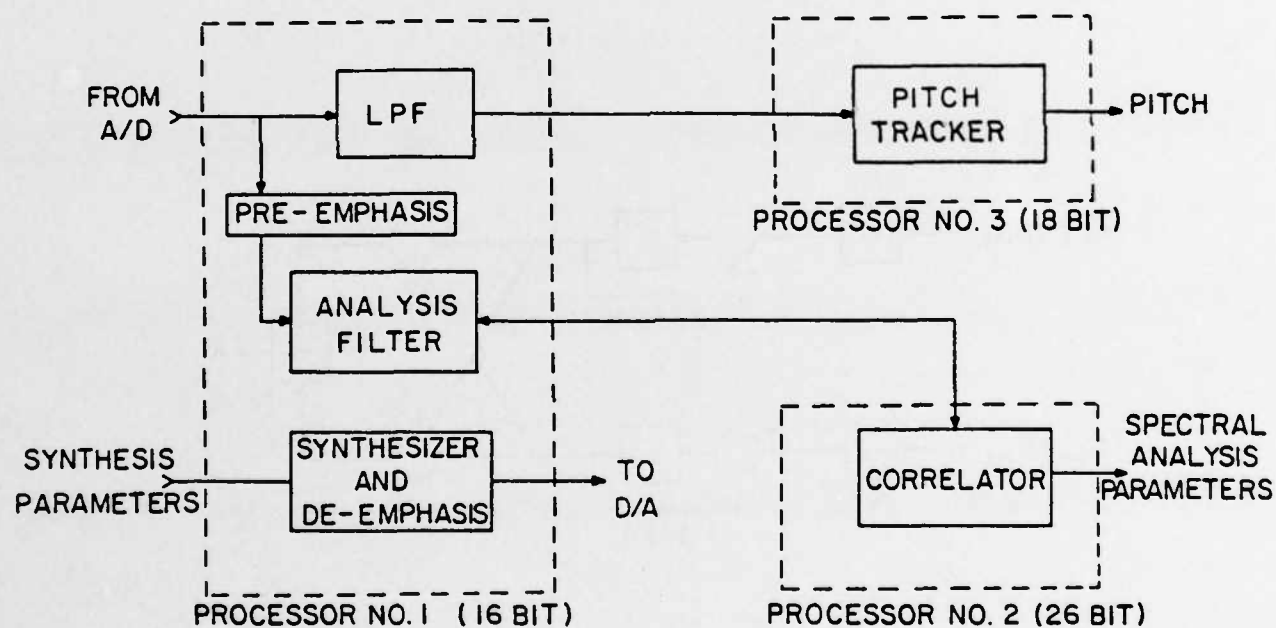


Fig.1 Signal flow chart for single-chip vocoder

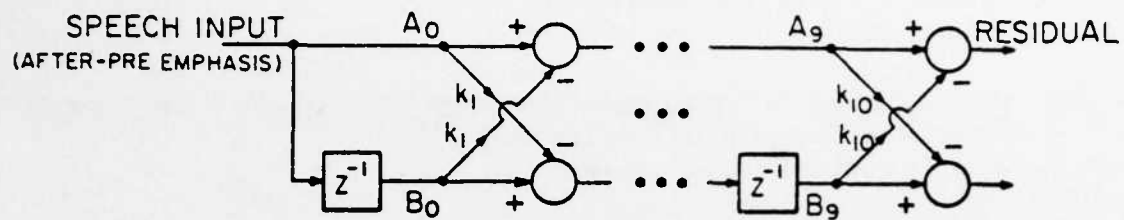


Fig.2 Flow chart for the analysis lattice filter

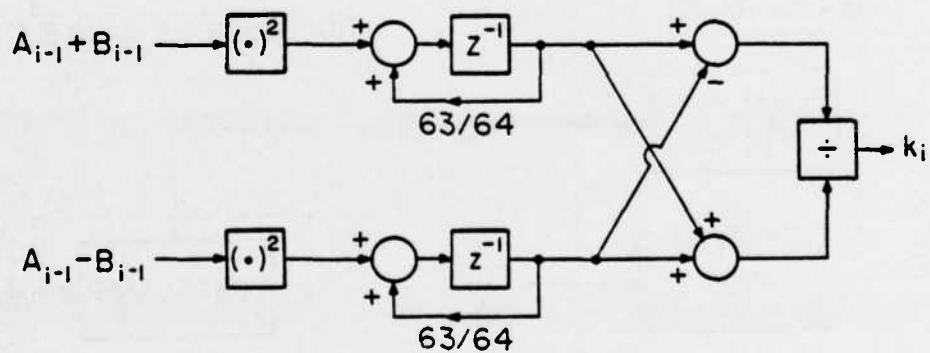
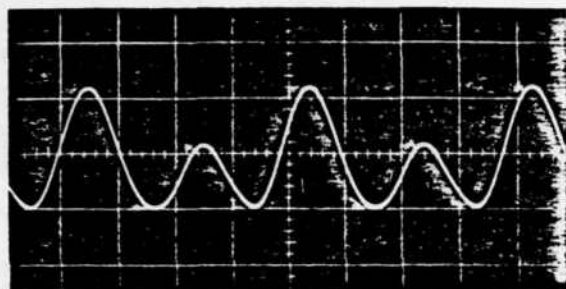
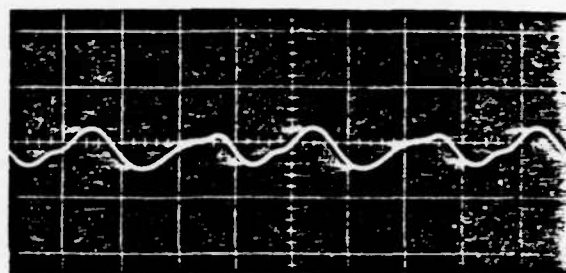


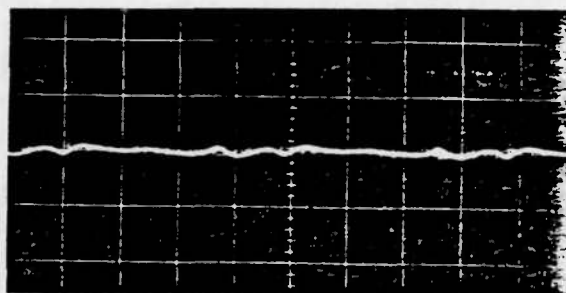
Fig.3 Flow chart for the correlator



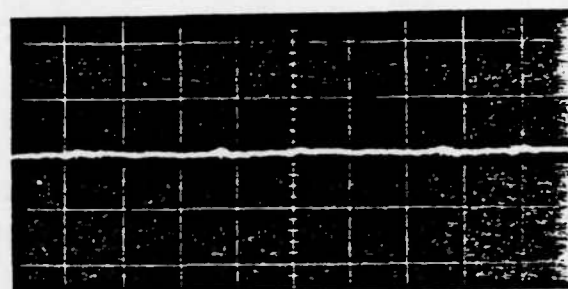
Input



Output of 1st stage



Output of 2nd stage



Output of 10th stage (residual)

Fig.4 Signals at different points in lattice analyzer  
(1ms/div horz; 2V/div vert)

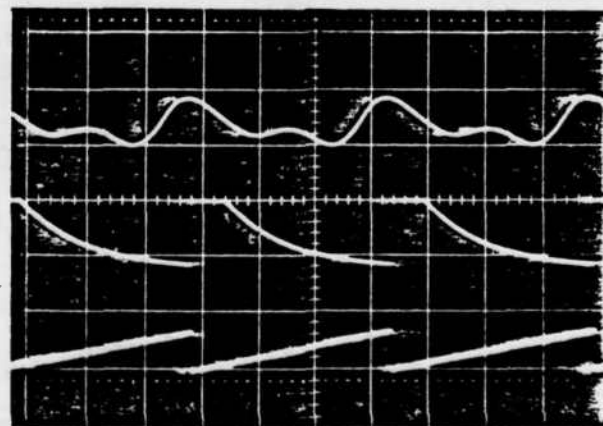


Fig.5 Pitch detector data (5ms/div horz)  
Top trace: input signal (5V/div vert)  
Middle trace: threshold waveform (1V/div vert)  
Bottom trace: pitch period counter (.5V/div vert)

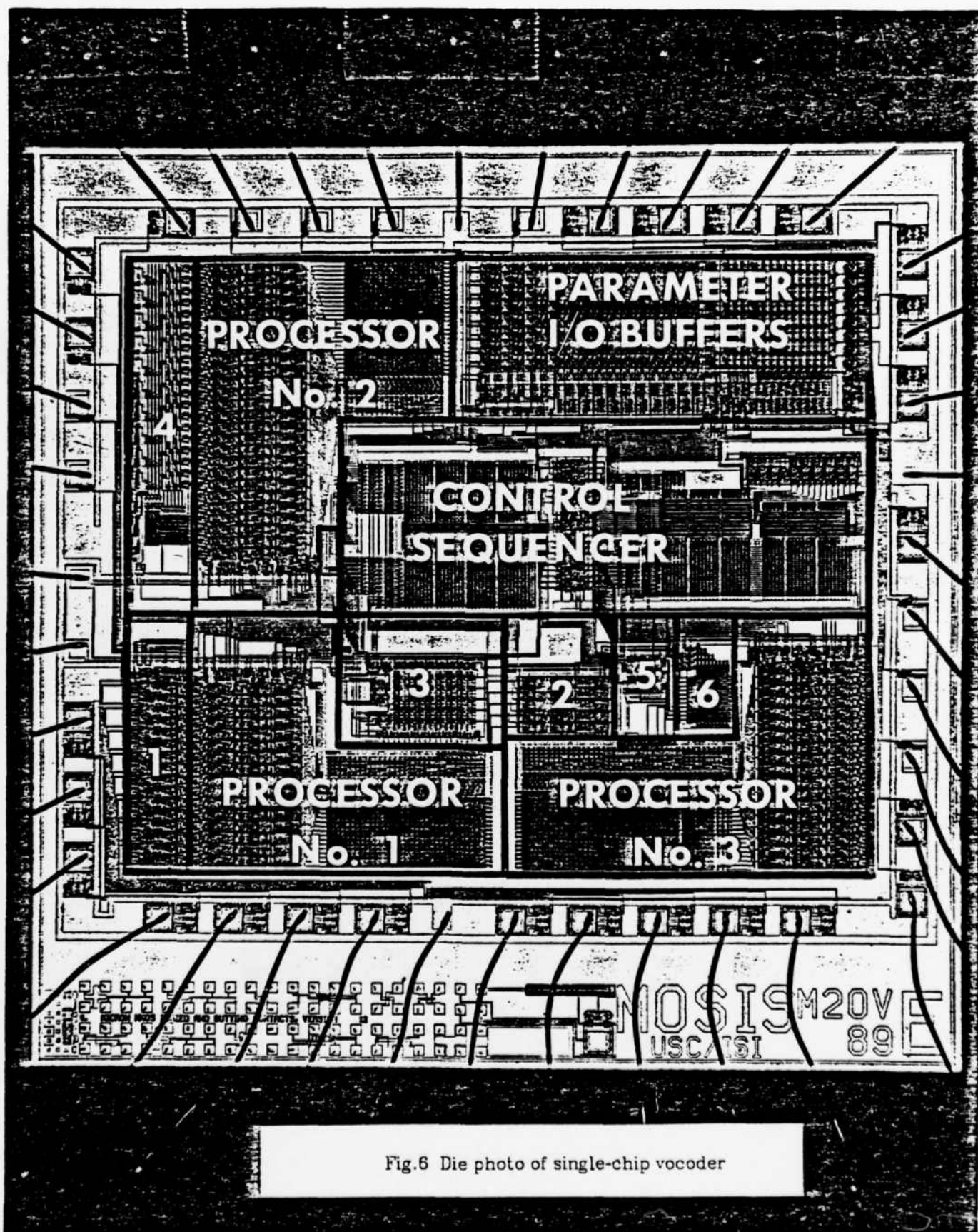


Fig.6 Die photo of single-chip vocoder



## A MULTIRATE ROOT LPC SPEECH SYNTHESIZER

Chia-Chuan Hsiao  
Codex Corporation, Mansfield, MA 02048

Robert W. Broderseu  
University of California, Berkeley, CA 94720

### ABSTRACT

The root LPC system has closed form analysis and formant like synthesis structure. By using quadratic coefficient quantization and section repeat its data rate can be lower than 1Kbps. By including representative residual signal of variable repetition rate its quality can be continuously improved. A special purpose NMOS-LSI chip was built to implement the synthesis function.

### INTRODUCTION

The root LPC is an extension of LPC and has a formant like structure. It has the closed form analysis as in LPC and can reach a very low data rate as in formant synthesis. The prediction residual signal is included in synthesis for better speech quality as in the RELP (Residual Excited Linear Prediction) system.[1] By properly processing the synthesis filter parameters and the residual signal, variable data rates with a wide range of speech quality can be obtained.

In LPC system the reflection coefficients have been named the "best" set for coding in terms of filter stability upon quantization and inherent ordering of the coefficients.[2]

#### Filter Stability

The roots of  $A(z)$ , the prediction polynomial, can also guarantee the stability of the synthesis filter  $1/A(z)$  by always residing inside the unit circle in the  $Z$ -plane.

#### Parameter Inherent Ordering

The inherent ordering of parameters is good for parameterizing signal since the parameter positions are also used for carrying information. But a good analyzer should be able to decompose the speech into basic features which are represented by unrelated parameters or groups of parameters. Then it would be straightforward to emphasize or de-emphasize any feature without side effects on others in reconstructing the speech. From this point of view the inherent ordering of all parameters is not really a good thing.

The roots of  $A(z)$ , like the formants, do not possess an inherent order. If according to some characteristic measurement the root pairs were put in order, the advantage of inherently ordered parameters could be regained.

### ROOT LPC SPEECH SYNTHESIS

The first work for stepping from straight LPC to root LPC is to find the roots of the LPC polynomial

$$A(z) = 1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_M z^{-M}$$

Research sponsored by DARPA under contract N00034-K-0251

$$= \prod_{i=1}^{M/2} (1 + b_i z^{-1} + c_i z^{-2})$$

where  $a_k$ 's,  $1 \leq k \leq M$ , the prediction coefficients, are all real. Bairstow's method is the most popular numerical method for finding the roots of a polynomial with real coefficients without using complex arithmetic.[3]

#### Filter Memory Effect at Frame Boundaries

In the root LPC synthesizer if the quadratic factors were not well ordered, some sections could have a large coefficient change from frame to frame. In this case the waveform discontinuities at frame boundaries may be serious.[4] An example of this memory effect is shown in Fig. 1.

In Fig. 1 there are two and a half frames of synthesized speech. Each frame contains 200 samples, which are equal to 2 pitch periods of speech. The filter coefficients for them are all the same except factors #1 and #3 are interchanged in the frame of samples 400-599.

To minimize this memory effect, the filter input should be relatively larger than its memory. In cascade form filter the input to the quadratic section is the output of the preceding section. Therefore the preceding section with a relatively lower decay rate is expected.

The ordering scheme based on the decay rate is only to minimize the memory effect, but sometimes it is still audible. Experiments showed that "zeroing the memory" at the frame boundaries did not introduce any audible discontinuities. Therefore the "memory zeroing" method was taken to fix the memory effect problem. And the factor ordering will be used exclusively for the quantization and repeat algorithm which will be discussed next.

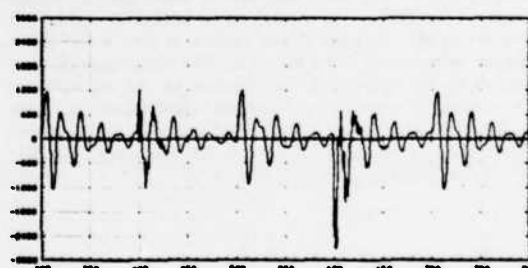


Figure 1: Memory Effect for Improper Factor Ordering

### DATA COMPRESSION

Parameter quantization and section repeat are used for data compression. Both of them depend heavily upon the ordering of the quadratic factors. A good factor ordering scheme makes the section functional variation across frame boundaries minimal so as to reduce memory effect and benefit parameter quantization and repeat.

### Pole Concentration Regions

Suppose the pole concentration regions for the quadratic sections are located already. The factors are ordered by maximizing the number of matches between the quadratic factors and the concentration regions [5]. A match means the roots of the factor are within the assigned region. The factor assigned to region #1 is put in section 1 of the cascade filter and so on.

The concentration regions can be derived by a training process, which may be replaced by a fixed assignment for simplicity. The region assignments for LPC-12 are (1)  $\{0.6 < r < 1, 0 < \theta < 0.8\}$ , (2)  $\{0.6 < r < 1, 0.5 < \theta < 1.6\}$ , (3)  $\{0.6 < r < 1, 0.8 < \theta < 2.0\}$ , (4)  $\{0.6 < r < 1, 1.5 < \theta < 2.4\}$ , (5)  $\{0.6 < r < 1, 1.8 < \theta < 3.14\}$ , and the real axis, (6)  $\{0.6 < r < 1, 2.6 < \theta < 3.14\}$ ,  $\{0 < r < 0.8, 0 < \theta < 3.14\}$ , and the real axis, where  $r$  is the radius and  $\theta$  is the angle from the real axis in the Z-plane. They have overlapped areas between neighboring regions. Only the regions for voiced frames are listed. The unvoiced frame using LPC-4 has only two factors. They do not need to be ordered by matching. Also only the regions above the real axis are specified here because all pole locations are symmetrical about the real axis.

In an experiment for 30.81 seconds of male and female speech, there are less than 3% unmatched sections for these region assignments.

### Quadratic Coefficient Quantization

A short code word is used for matched sections and extra bits are used for the unmatched ones. Suppose there are  $N_i$  bits allocated to the short code word for the  $i$ -th section. One code is reserved for the unmatched flag. Then  $2^{N_i} - 1$  code points are distributed in the  $i$ -th section region. For unmatched sections, a long code word is indicated by all 1's in these  $N_i$  bits, and the following  $M_i$  more bits are used for coding the unmatched coefficients. Since the unmatched sections happened only with low probability, the average number of bits for the  $i$ -th section is only slightly more than  $N_i$ .

The number of bits used are listed below, where  $i=7,8$  are for the unvoiced frames [6].

$i$	1	2	3	4	5	6	7	8
$N_i$	6	7	4	3	4	4	4	4
$M_i$	6	6	6	7	7	7	2	5
# codes	127	191	79	135	143	143	19	47

### Quadratic Section Repeat

In root LPC parameter repeat can be applied to individual sections instead of the whole parameter set. Suppose there are  $N$  frames and  $N$  sets of quadratic coefficients available for some specific section. The quadratic coefficient sets  $Q_i$  is derived by analyzing the speech waveform in frame # $i$ . In Figs. 2 the horizontal axis is the frame number and the vertical axis is for the  $Q_i$ 's. The main diagonal from the bottom left to the upper right corresponds to the case without repeat. An example of some forward and backward repeats are plotted in Fig. 2, where  $Q_1$  is used for frames #1 and #2,  $Q_4$  for frames #4 and #5 and  $Q_8$  for frames #7, #8 and #9. Thus only 7 sets of parameters are needed for 11 frames in this case.

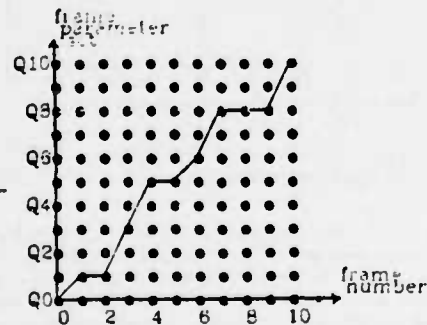


Figure 2: Coefficient assignment with section repeat

The goal of section repeat is to use minimum number of  $Q_i$ 's for all frames under some quality deviation allowance. The quality deviations for all points on the graph can be called the local errors. It is assumed that  $Q_i$  is the best set for the  $i$ -th frame and without quality deviation. Then for all points on the main diagonal the local errors are zero. For off-diagonal points  $(i, j)$ ,  $i \neq j$ , which means  $Q_i$  replaces  $Q_j$  for the  $i$ -th frame, some quality deviation is introduced. All points on a column with local errors less than a threshold indicate the associated  $Q_i$ 's could be used in that frame under allowable quality deviation.

The extra bits needed for updating the coefficients is called the transition cost. The transition cost is zero if no updating is needed, which corresponds to staying on the same row for the next column. To synthesize the whole speech with minimum number of  $Q_i$ 's means a path from the left to the right on the graph with minimum number of rows. The dynamic programming technique can be used to search for the optimal path [7].

### Dynamic Programming

For each column on the frame-parameter plot, calculate the local errors for all points. If the local error is greater than the threshold, any path passing this point is inhibited, otherwise the point is available to all candidate paths to pass through.

For each path, its cost and error are defined as

$$\text{path cost} = \sum_{\text{path}} (\text{transition cost})$$

$$\text{path error} = \sum_{\text{path}} (\text{local error})$$

There could be many paths from the preceding column to each available point. Only that path with minimum cost needs to be saved so if more than one path has the same cost, the one with minimum path error is saved.

This procedure is repeated for all columns from the left to the right and the path terminated on the last column with minimum cost or error is taken. Then by back tracking from the right to the left the best scheme for section repeat is obtained.

Different thresholds can be set for the sections. Lower thresholds were set for the formant sections and higher thresholds for the non-formant sections.

## SPEECH QUALITY ENHANCEMENT

While the unvoiced sounds are important for syllable identification, the vowel sounds dominate the speech quality. To enhance the speech quality without paying the full price in bandwidth, only the residual signal for voiced sound is stored or transmitted.

### Representative Residual for Voiced Frames

The single pulse in the pulse train for the excitation for the voiced sound could be called the "anchor pulse". In addition to the anchor pulse, the residual signal has many "residual pulses" for adjusting local prediction error with the original waveform. There are two ways in using part of the residual pulses. One is to add the most effective pulses first [8]. Another way is to add them one right after the other from the anchor pulse until the next one.

Experiments showed that the residual pulses must be dense and long enough for speech quality enhancement. For partial quality enhancement a representative residual of a full pitch period is used for some continuous frames. The highest quality with highest data rate needs one residual per frame.

### Target Waveform

For each frame one pitch period of representative residual pulses is used for quality enhancement. The original speech waveform of one pitch period which would be regenerated with the residual pulses is called the "target waveform". The rest of the waveform in a frame will be thought as replicas of the target waveform with minor deviation.

tions. Experiments showed the speech with all voiced frames filled with their target waveforms sound close to the original.

The target waveform is located by cross correlating the impulse response of the synthesis filter and the pre-emphasized original waveform. The waveform segment having the maximum cross correlation is taken as the target waveform.

#### Derivation of Representative Residual

If a representative residual for  $N$  frames is desired, there are  $N$  target waveforms to be regenerated with  $N$  sets of parameters and gain but only one residual. Each residual pulse will be used for adjusting  $N$  local prediction errors.

Suppose the  $j$ -th sample of the  $i$ -th target waveform  $e_{ij}$  is predicted from the preceding samples as  $e_{ij}'$  and adjusted by the residual pulse  $g_i r_j$ , where  $g_i$  is the gain factor for frame # $i$ . then their error energy  $E_j$  is

$$E_j = \sum_{i=0}^{N-1} e_{ij}^2 = \sum_{i=0}^{N-1} (e_{ij} - e_{ij}' - g_i r_j)^2.$$

To find the minimum of  $E_j$  by varying  $r_j$ , its derivative is set to zero.

$$\frac{dE_j}{dr_j} = -2 \sum_{i=0}^{N-1} g_i (e_{ij} - e_{ij}' - g_i r_j) = 0.$$

$$r_j = \frac{\sum_{i=0}^{N-1} g_i (e_{ij} - e_{ij}')}{\sum_{i=0}^{N-1} g_i^2}.$$

This procedure is repeated from the anchor pulse for all residual pulses.

The algorithm developed above for the representative residual pulse is good for any value of  $N$ , the number of frames. The larger  $N$  is, the less there is the ability of local error adjustment by the residual pulse. Therefore when the residual repetition rate,  $1/N$ , increases the synthesized speech quality also increases.

### THE MULTIRATE ROOT LPC SYSTEM

The multirate root LPC analyzer and synthesizer are depicted in Figs. 3 and 4. The multirate capability is accomplished with three independent processes:

1. quantization and its code books,
2. section repeat and its thresholds, and
3. representative residual and its repetition rate.

A subjective speech quality test and a diagnostic rhyme test for the syntheses of different data rates were taken by ten people to check the quality and intelligibility variation.[9] The subjective quality test consists of five sentences spoken by male and female speakers with wide range of pitch period. These sentences are

1. This is a demonstration of synthetic speech. (male)
2. Please fasten your seat belt. (female)
3. Dr. Bob is on vacation again. (male)
4. Did you exercise in the last hour? (female)
5. Synthetic speech may be useful in many consumer applications. (male)

For each sentence, five different syntheses were made and played pairwise for quality comparison. These five syntheses, all of which were automatically performed, are: (1) Quantized root LPC with section repeat, (2) Quantized root LPC, (3) Root LPC, (4) Root LPC with one representative residual per five voiced frames, and (5) root LPC with one representative residual per two voiced frames. The listeners were asked to make one decision from (a) apparently lower, (b) a little lower, (c) the same, (d) a little higher, and (e) apparently higher in quality for the second synthesis compared to the first one of each pair. The scores for the relative quality are -2, -1, 0, +1, and +2 for these answers respectively.

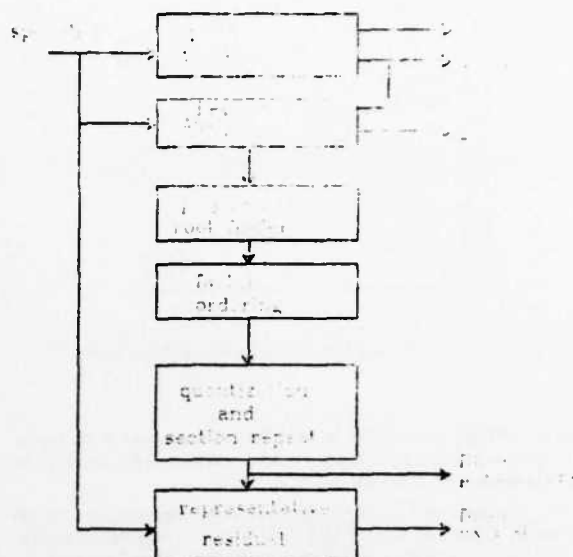


Figure 3: Multirate Root LPC analyzer System

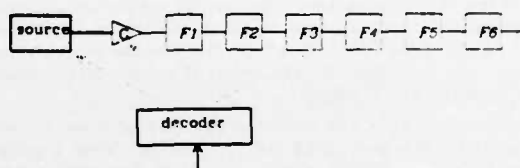


Figure 4: Multirate Root LPC Synthesizer System

The average data rates for the above five syntheses are 964, 1443, 6294, 8577, 10930 bits per second respectively for the above five sentences. The subjective quality test results are shown in table and Fig. 5.

synth. pair	lower	a little lower	the same	a little higher	higher	total score
	-2	-1	0	+1	+2	
1-2	0	0	25	19	6	31
2-3	0	2	20	23	5	31
3-4	0	12	23	14	1	4
4-5	1	5	32	12	0	5

In the plot the vertical scale is the overall relative quality scores from ten people. The plot showed that many people felt the synthesized speech quality is increased with the data rate.

The results of the diagnostic rhyme test for the five data rates are listed below.

Synthesis	1	2	3	4	5	original
bps	964	1443	6294	8577	10.93K	120K
DRT	85%	88%	90%	93%	90%	99%

There is only a little intelligibility loss for decreasing the data rate.

### CHIP OPERATION AND ARCHITECTURE

The implemented synthesizer chip is basically a linear time-varying cascade quadratic all-pole filter. It has programmable filter coefficients. The excitation received will be filtered with the stored coefficients and then sent out with one clock cycle delay. The filter coefficient and excitation inputs and the speech sample output are all through the same parallel I/O port. Thus a line to differentiate the input data as being coefficients or excitation is required. Also a first-



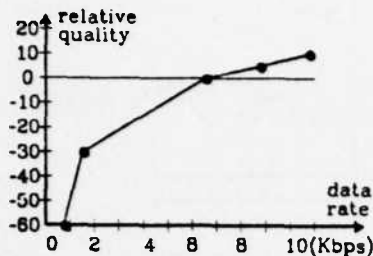


Figure 5: Synthesized Speech Quality versus Data Rate

in-first-out (FIFO) data buffer is used for asynchronous data input. This chip is controlled by a master microprocessor iAPX186.[10] the system block diagram is shown in Fig. 6.

The synthesizer is triggered to start processing for one sample by an inverted pulse on the START pin. First it reads in the excitation, multiplies it with gain factor, and de-emphasizes and outputs the speech sample from last clock cycle. Then the excitation is filtered with the stored coefficients and memory state values. If the UPDATE is high voltage, the updating gain factor and 12 coefficients are read in through the data port to overwrite the stored values. The filter memory is cleared if the UPDATE is high. After all the six quadratic sections had been performed, the synthesizer responds a STOP pulse and waits for next START pulse.

The IN and OUT are used to synchronize data input and output through the 18-bit data port. DAV indicates whether datum is available in the FIFO. For NMOS circuits, 5V voltage is required across VDD and GND. A nonoverlapped two phase clock is applied to Ph1 and Ph2.

Internally the chip consists of an 18-bit signal processor block,[11] 32-word RAM, program counter, 24 bits X 96 words program ROM, RAM address indexing and a random logic circuit.

#### SUMMARY

A root LPC speech synthesizer with a variable bit rate has been built using a special purpose NMOS-LSI circuit. The data rate can be from less than 1Kbps to greater than 10Kbps and any value in between. The output speech quality is determined by the desired input data rate.

The root LPC speech synthesis is a combination of formant and LPC systems. The LPC polynomial is factored into quadratic factors. These factors are ordered by optimally matching their pole locations with overlapped regions in the Z-plane. The ordered factors are then used as the stages of a cascade form formant synthesizer. The memory effect is fixed by zeroing the memory at frame boundaries.

To lower the bit rate without losing the intelligibility, quadratic coefficient quantization based on a perceptual measurement and a section repeat algorithm based on dynamic programming are used. To upgrade the speech quality in terms of naturalness and smoothness, representative prediction residuals are used as the synthesis filter excitation for the voiced frames. By adjusting the maximal allowable spectral distances for section repeat and/or the repetition rate of the representative residuals, a continuum of bit rates with varying speech quality can be achieved.

A chip was designed to implement this synthesizer. It uses a macrocell design approach and requires 16.2 square millimeters of die area. For 10kHz sampling rate it runs with a master clock of 3.4MHz.

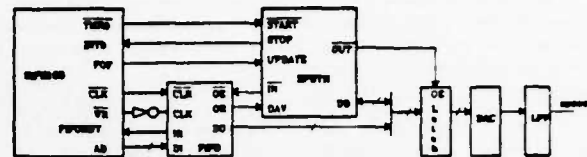


Figure 6: System Block Diagram of the Synthesizer

#### REFERENCES

1. H. Katterfeldt, "A DFT-Based Residual-Excited Linear Predictive Coder (RELPC) for 4.8 and 9.6 kb/s," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 824-827 (1981).
2. R. Viswanathan and J. Makhoul, "Quantization Properties of Transmission Parameters in Linear Predictive Systems," *IEEE Trans. Acoust., Speech, and Signal Process.* ASSP-23 pp. 309-321 (June 1975).
3. A. Ralston, *A First Course in Numerical Analysis*, McGraw-Hill Book Co., New York (1965).
4. D. Vetter, J. Stork, K. Skoge, and P. Ahrens, "LPC Speech I.C. Using a 12-pole Cascade Digital Filter," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 467-470 (1981).
5. E. U. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, N. Y. (1976).
6. N. Morgan, "Perceptually-Based Coding for Root LPC Synthesis," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing, Paris*, pp. 944-946 (1982).
7. R. E. Bellman, *Dynamic Programming*, Princeton Univ. Press, Princeton, N.J. (1957).
8. B. S. Atal and R. Remde, "A New Model of LPC Excitation for Producing natural-Sounding Speech at Low Bit Rates," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 614-61 (1982).
9. W. D. Voiers, "Diagnostic Evaluation of Speech Intelligibility," in *Speech Intelligibility and Speaker Recognition*, ed. M. E. Hawley, Dowden, Hutchinson, and Ross, Stroudsburg (1977).
10. iAPX 86/88, 186/188 User's Manual, Hardware Reference, Intel, Santa Clara, CA 95051 (May, 1983).
11. S. Bjorn, *One Year at the University of California, Berkeley, USA. Designing and Fabricating Integrated Circuits*, Norwegian Defense Research Establishment (NDRE), Forsvarets Forskningsinstitutt (FFI), Post Office Box 25, N-2007 Kjeller, Norway (FFI/RAPPORT-83/7003, March 1, 1983).

PUBLICATIONS ON TECHNOLOGY

## TECHNOLOGY

The following section contains papers and reports relating to research in computer Technology. They describe work which was wholly or in part performed under the sponsorship of the DARPA grant.

- (1) C. Hu (invited paper), "Hot-Electron Effects in MOSFET's," *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington, D.C., December 1983, pp 176-181.
- (2) S. Tam, C. Hu, "Hot Electron Induced Photo-Carrier Generation in Silicon MOSFET's", accepted for publication in *IEEE Transactions on Electron Devices*.
- (3) T.C. Ong, K.Y. Terrill, S. Tam, C. Hu, "Photo Generation in Forward-biased Silicon PN Junctions," *IEEE Electron Device Letters*, EDL-4, No. 12, December 1983, pp 460-462.
- (4) S. Tam, P.K. Ko, C. Hu, "Lucky-Electron Model of Channel Hot Electron Injection in MOSFET's," accepted for publication in *IEEE Transaction in Electron Devices*.
- (5) M.S. Liang, C. Chang, W. Yang, C. Hu, R.W. Brodersen, "Hot-Carriers Induced Degradation in Thin Gate Oxide MOSFET's" *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington, D.C., December 1983, pp 194-197.
- (6) C. Chang, M.S. Liang, C. Hu, R.W. Brodersen, "Carrier-Tunneling Related Phenomena in Thin Oxide MOSFET's" *Technical Digest of 1983 IEEE International Electron Devices Meeting*, Washington, D.C., December 1983, pp 194-197.
- (7) S. Holland, I.C. Chen, T.P. Ma, C. Hu, "On Physical Models for Gate Oxide Breakdown," submitted to *IEEE Electron Device Letters*.
- (8) B.J. Sheu, C. Hu, "Modeling the Switch-Induced Error Voltage on a Switched Capacitor," *IEEE Transactions on Circuits and Systems*, CAS-30, No. 12, December 1983, pp 911-913.
- (9) B.J. Sheu, C. Hu, "Switch-Induced Error Voltage on a Switched Capacitor," accepted for publication in *IEEE Journal of Solid State Circuits*.
- (10) K. Terrill, C. Hu, "Substrate Potential Calculation for Latch-up Modeling," accepted for publication *IEEE Electron Device Letters*.
- (11) K. Terrill, C. Hu, A. Neureuther, "Computer Analysis on the Collection of Alpha-Generated Charge for Reflecting and Absorbing Surface Conditions around the Collector," *Solid State Electronics*, No. 1, January 1984, pp. 45-52.

## Hot-Electron Effects in MOSFET's

(Invited Paper)

Chenming Hu

Department of Electrical Engineering  
and Computer Sciences

University of California, Berkeley, CA 94720

## Abstract

The physics of several hot-electron currents and their impact on IC performance and reliability is reviewed.  $V_d - V_{dsat}$  is emphasized as the driving force of all hot-electron effects.  $V_g$ ,  $V_{sub}$ , and  $L$  affect the hot-electron effects only through their influence on  $V_{dsat}$ . A simple hot-electron scaling rule is to scale  $V_d$ ,  $V_t$ , and  $\sqrt{x_{ox}x_j}$  in proportion to  $L$ . Several proposed structural changes should provide considerable relief to the hot-electron problem.

## Introduction

1974 saw the introduction of 5V 1K static RAMs having  $6\mu\text{m}$  channels and 1200A gate oxide. Today's 5V RAMs often have channel lengths shorter than  $1.5\mu\text{m}$  and gate oxides thinner than 250A. The power supply voltage has escaped scaling for reasons of compatibility with existing systems and circuit speed and margins. Voltage are even higher in bootstrapped circuits and special high voltage circuits such as EPROMs. Even after relief comes in the form of lower power supply voltages, the same reasons cited above plus the imperfect control of threshold voltages and the nonscalability of subthreshold IV characteristics will continue to peg the power supply voltage near unacceptable values.

As a result, the hot-electron effects have caused concerns for and actual occurrences of circuit failures, performance losses, and reliability problems. This paper attempts to clarify the conceptual models of the effects and suggests some rules and tools that may simplify the visualizing, characterization, and scaling of the hot-electron effects. It is not an exhaustive or balanced survey of the literature on this subject. The discussion will be devoted to N-channel MOSFETs. The hot-carrier effects are weak enough in PMOS not to cause immediate concern.

## Hot-Electron Currents and Their Impact

The term "hot-electron effects" often refers only to the phenomenon of device degradation due to channel hot-electron injection [1]. A clearer and more useful picture may be obtained by defining it as all the hot-electron currents described below and their impact on circuit performance and reliability.

Referring to Fig. 1, the substrate current,  $I_{sub}$ , results from the hole generation by the channel hot electrons through impact ionization. If the impact ionization coefficient is  $A_i \exp(-B_i/E)$ , where  $E$  is the electric field,  $I_{sub}$  can be shown to be [2]

$$I_{sub} \approx C_1 I_d e^{-\frac{B_i}{E_m}} \approx 2 I_d e^{-\frac{1.7 \times 10^6}{E_m}} \quad (1)$$

$E_m$  in V/cm is the maximum channel electric field, i.e., the field at the drain end of the channel. Excessive  $I_{sub}$  can overload on-chip substrate-bias generators. The potential (ohmic voltage) variations in the substrate produced by the flow of  $I_{sub}$  can cause  $V_t$  variations, and, in severe cases, snap-back (avalanche) breakdown of the MOSFET [3] or latch-up in CMOS circuits [4].

The gate current,  $I_g$ , has a similar theoretical dependence on  $E_m$  (V/cm) [2],

$$I_g \approx C_2(E_{ox}) I_d e^{-\frac{\phi_b}{\lambda E_m}} \quad V_g > V_d \quad (2)$$

$E_{ox} = (V_g - V_d)/x_{ox}$  is the oxide field near the drain where  $C_2(E_{ox})$  increases from about  $10^{-3}$  at  $E_{ox} \approx 0$  to about  $4 \times 10^{-3}$  at  $E_{ox} = 10^6$  V/cm.  $\lambda$  is the channel hot-electron mean-free-path and is about 78A [5],  $\phi_b$  is the Si/SiO<sub>2</sub> barrier height in volts modified by barrier lowering [6].  $\phi_b$  is about 3.1V at  $E_{ox} \approx 0$  and about 2.5V at  $E_{ox} = 10^6$  V/cm. The exponential term is the probability for an electron to gain more energy than  $q\phi_b$  without suffering a collision.  $C_2$  is the probability for an energetic electron to be injected into the oxide. Some other combinations of  $\lambda$  and  $C_2$  values can also fit the  $I_g$  data well. The injected electrons may be trapped in the oxide,

causing  $V_t$  drift, or generate interface traps, causing degradations in electron mobility and subthreshold characteristics, and additional  $V_t$  shift [7].

Minority carrier (electron) current,  $I_n$ , is known to be released into the substrate from a MOSFET operating in the saturation region. These electrons were mistakenly believed to be the product of secondary impact ionization [4], which is in fact many orders of magnitude too small to explain  $I_n$  [8]. Two mechanisms are responsible as indicated in Fig. 1. When  $I_{sub}$  is low, the dominating mechanism is photo-carrier generation where the photons are produced by the hot channel electrons [9]. Bremsstrahlung is believed to be the photon generation mechanism and the probability for a channel electron to generate a photon with energy  $h\nu$  is proportional to  $\exp\left[-\frac{h\nu}{q\lambda E_m}\right]$ . The total rate of photon generation translate into [10]

$$I_n \approx 6 \times 10^{-5} I_d e^{\frac{-1.3eV}{q\lambda E_m}} = 6 \times 10^{-5} I_d e^{\frac{-1.65 \times 10^5}{E_m}} \quad (3)$$

where 1.3eV may be considered as the average energy of the photon spectrum. If the substrate resistivity is high and at large enough  $I_{sub}$ , the source-substrate junction may be sufficiently forward biased such that  $I_n$  is dominated by the electron injection from the source [8,11]. This second mechanism is also depicted in Fig. 1. Once the electrons are deposited in the substrate, by either mechanism, they may be collected by nearby nodes as excess leakage currents. These leakage currents,  $I_{coll}$  in Fig. 1, are known to cause DRAM refresh-time degradation [8] and can discharge other charge-storage nodes or even low-current carrying nodes [4]. A portion of the photon spectrum has large penetration depth in Si.  $I_{coll}$ , therefore, varies with the separation between the collecting node and the culprit MOSFET in accordance with a long effective decay length of about 800 $\mu m$  (when photo-carrier generation is the dominating mechanism) as shown in Fig. 2.

Hot electrons also indirectly affect the IV characteristics. When  $I_{sub} R_{sub} \geq V_{sub} + 0.6V$ , significant current,  $I_e$  in Fig. 1, may be injected from the source to the drain, in addition to the gate-controlled surface current, causing the familiar rise in the I-V curve [11]. When the condition  $M$  (multiplication factor)  $> \frac{1}{\alpha_{nps}}$  is also satisfied, often at a yet higher  $V_d$ , snap-back breakdown occurs [3].

### Electron Temperature versus Field

Assuming an isotropic hot-electron gas at temperature  $T_e$ , one expects the gate current to be proportional to  $\exp(-q\phi_b/kT_e)$  [5,12]. By fitting the  $I_g$  data to numerically simulated [12,13] or analytically modeled [5] channel electric field, these studies have independently concluded that

$$T_e \text{ (Kelvin)} \approx 7.5 \times 10^{-3} E \text{ (V/cm)} \quad (4)$$

With this relationship the gate current essentially has the form  $I_g = C_3 \exp(-\phi_b/\beta E_m)$ , similar to Eq. 2.

### Correlations Among the Hot-Electron Currents

By eliminating  $E_m$  from any pair of Eqs. 1, 2, and 3, a simple power-law correlation can be obtained. For example, substituting Eq. 1 for  $E_m$  in Eqs. 2 and 3 yields

$$\frac{I_g}{I_d} \approx C_2 \left( \frac{I_{sub}}{C_1 I_d} \right)^{\frac{\phi_b}{\lambda B_1}} = 0.2 C_2 \left( \frac{I_{sub}}{I_d} \right)^{\frac{\phi_b}{1.3}} \quad (5)$$

$$I_n \approx 3 \times 10^{-5} I_{sub} \quad (6)$$

Fig. 3 demonstrates the correlation between  $I_g$  and  $I_{sub}$ . With this correlation we may monitor  $I_g$  by measuring the much larger  $I_{sub}$ . The correlation in Eq. 6 has already been reported in the literature without explanation [4]. In addition,  $I_{sub}$  is linearly proportional to  $I_e$ , or  $I_d - I_{dsat}$  [11]. It is important to note that these correlations are independent of device dimensions and bias voltages. Recently the  $I_g - I_{sub}$  correlation was shown to apply to even a 0.14 $\mu m$  channel device [14].

### Simple Model of Channel Field

2D and 3D simulations are the most reliable means of evaluating the channel electric field [12,15]. The MINIMOS program [15] is freely distributed and contains an  $I_{sub}$  model. No simulation program available to the public contains an  $I_g$  model. While most analytical field models were not accurate enough for use in hot-electron studies, a recent quasi-analytical model has succeeded remarkably well [5]. It is based on a quasi-2D approach [16]. A much simplified version is [5,3]

$$E_m = \frac{(V_d - V_{dsat})}{K} \approx \frac{(V_d - V_{dsat})}{\sqrt{3x_{ox}x_j}} \quad (7)$$



where  $x_j$  is the junction depth and  $x_{ox}$  is the oxide thickness. The drain saturation voltage is approximately [17]

$$V_{dsat} = \frac{(V_g - V_t)LE_{sat}}{V_g - V_t + LE_{sat}} \quad (8)$$

where  $L$  is the effective channel length in  $\mu m$ , and  $E_{sat}$ , the critical field for velocity saturation is about  $3 \times 10^4$  V/cm for lightly doped substrates and larger for scaled devices in heavily doped substrates.

Eqs. 7 and 8 may not be accurate enough to predict hot-electron currents, but should be useful in predicting the trend and understanding the dependence of the hot-electron effects on bias and dimensions as shown below.

#### Dependence on Bias Voltage

All bias voltage dependence is contained in Eqs. 7 and 8. Specifically,  $V_g$  and  $V_{sub}$  affect  $E_m$  only through  $V_{dsat}$ . Sing and Sudlow [18] have arrived at an equation similar to Eq. 7. One of their figures is reproduced in Fig. 4. The initial rise in the familiar bell-shaped curve is due to rising  $I_d$  (see Eq. 1) and the eventual fall is due to falling  $E_m$  (see Eqs. 7, 8). Peak  $I_{sub}$  has been fitted to  $\exp(-\alpha/V_d)$  with a dependence on  $L$  [12].  $I_g$  also exhibits a bell-shaped curve [19] peaking at  $V_g \approx V_d$ . For  $V_g > V_d$ ,  $I_g$  falls with increasing  $V_g$  because of falling  $E_m$ . For  $V_g < V_d$ ,  $I_g$  decreases with decreasing  $V_g$  and is almost independent of  $V_d$ . In this case, the channel may be divided into two parts at the point where the channel potential is equal to  $V_g$ . Between this point and the drain, channel hot-electron injection is negligible because of the retarding oxide field. Between this point and source the channel field, hence  $I_g$ , is independent of  $V_d$  and decreases with decreasing  $V_g$ . The maximum  $I_g$  has been fitted to  $\exp(bV_d)$  with  $b$  dependent on  $L$  [12,19].

#### Concept of Critical Field

In the theory of pn junction breakdown, a helpful concept is that junction breakdown occurs when the peak electric field exceeds a certain critical value. A similar concept can be introduced for the hot-electron effects. According to Eqs. 1, 2, and 3, when  $E_m = 1.5 \times 10^5$  V/cm,  $I_{gmax}$  is about  $10^{-15}I_d$ , a common criterion for acceptable long-term stability [1,12];  $I_{sub} \approx 2 \times 10^{-5}I_d$  and  $I_n \approx 6 \times 10^{-10}I_d$  are also quite acceptable. Therefore, we might

remember  $1.5 \times 10^5$  V/cm as the critical field,  $E_c$ , for hot-electron effects.

#### Device Size and Voltage Limits and Scaling Rule

Perhaps surprisingly, channel length affects  $E_m$  only indirectly, through  $V_{dsat}$  (Eqs. 7 and 8). In order to ensure that  $E_m$  is less than the critical field,  $E_c$ ,  $V_d - V_{dsat}$  must be smaller than  $KE_c$  (see Eq. 7). This leads to

$$V_d < \frac{1}{2} \{ KE_c + V_t + \left[ (KE_c - V_t)^2 + 4E_{sat}L(KE_c - V_t) \right]^{1/2} \} \quad (9)$$

$$\stackrel{L \rightarrow 0}{=} KE_c$$

Eq. 9 is plotted (using  $E_{sat} = 5 \times 10^5$  V/cm) in Fig. 5 together with measured voltages at which  $I_g \approx 10^{-15}I_d$  [12]. Reasonable agreement is obtained. The three  $KE_c$  values are in about the same ratio as the square roots of the oxide thickness as predicted by Eq. 7. Using  $x_j = 0.3 \mu m$ ,  $E_c$  is found to be about  $1.5 \times 10^5$  V/cm in close agreement with the expected critical field strength.

Once  $E_c$  and  $L$  are fixed, the only other design changes that can increase the  $V_d$  limit is to increase  $\sqrt{x_{ox}x_j}$ . This, of course, would degrade other device performances. Eq. 9 suggests a simple scaling rule for keeping all the hot-electron effects in check (keeping  $E_m$  below  $E_c$ ):  $V_d$ ,  $V_t$ , and  $\sqrt{x_{ox}x_j}$  all be scaled linearly with  $L$ .

#### Mechanism of Device Degradations

The link between hot-electron injections and device degradations is still poorly understood. What are the microscopic reactions or the macroscopic models of the degradation mechanisms? What governs or is the kinetics of degradation? What is the process dependence? Definite answers are not available. Fortunately, for the purpose of technology development, one may well concentrate on the reduction of  $E_m$  because the electron-trapping (or interface-traps-generation) efficiency usually does not vary as much as the hot-electron population, which can be greatly suppressed by reducing  $E_m$ . It is for this reason that a critical field can be chosen.

### Improved Structures

Many structures have been proposed [12,20] for expanding the device-size/voltage limits. There are two approaches. The buried-channel structure mainly moves the source of hot electrons farther away from the oxide. The lightly-doped drain (offset-gate) structure, As-P double implants, and phosphorus junctions chiefly reduce  $E_m$  in Eq. 7 by dropping some voltage in a buffer zone. For example, the lightly-doped drain structure shown in Fig. 6 [20] is optimized when the field is about constant ( $E_m$ ) throughout the length of the  $N^-$  region,  $L_N$ . Thus, the voltage capability of the device can potentially be improved by  $E_c L_N$  or about 1.5V per  $\mu m$  of  $L_N$ .

The degradations of other device characteristics due to some of these structural changes, if any, are quite acceptable. These structures should gain popularity with VLSI circuits or high voltage circuits. As-P double implants can be added to an existing process most easily. The lightly-doped drain structure provides the greatest flexibility in design trade-offs and for that reason may be favored in many cases.

### Acknowledgement

It is a pleasure to acknowledge the contributions of Drs. Ping Ko, Fu-Chieh Hsu, and Simon Tam, whose outstanding thesis research at the University of California provided some of the data presented here and shaped my understanding of the subject. I thank Simon for preparing some of the figures. This research is supported by the DARPA under Contract N00039-81-K0251.

### References

- [1] T.H. Ning et al., IEEE Trans. Electron Dev., ED-26, p.246, 1979.
- [2] S. Tam et al., IEEE Trans. Electron Dev., ED-29, p.1740, 1982.
- [3] F. Hsu et al., IEEE Trans. Electron Dev., ED-29, p.1735, 1982.
- [4] J. Matsunaga et al., IEDM Tech. Dig., p.736, 1980.
- [5] P.K. Ko et al., IEDM Tech. Dig., p.600, 1981. P.K. Ko, Ph.D. Thesis, University of Calif., 1982.
- [6] T.H. Ning et al., J. Applied Physics, vol.48, p.286, 1977.
- [7] E. Takeda et al., IEEE Trans. Electron Dev., p.675, 1983.
- [8] B. Eitan et al., IEEE Trans. Electron Dev., ED-28, p.1515, 1981.
- [9] P.A. Childs et al., Electronics Letters, vol.17, p.281, 1981.
- [10] S. Tam, submitted to IEEE Trans. Electron Dev.
- [11] F. Hsu et al., IEEE Trans. Electron Dev., ED-30, p.571, 1983.
- [12] E. Takeda et al., IEEE Trans. Electron Dev., ED-29, p.611, 1982.
- [13] A. Ito et al., Proc. Int. Reliability Physics Conf., p.96, 1983.
- [14] S. Tam et al., IEEE Electron Device Letters, EDL-4, p.249, 1983.
- [15] A. Schutz et al., Solid-State Electr., vol.25, p.177, 1982.
- [16] N.A. ElMansy and A.R. Boothroyd, IEEE Trans. Electron Dev., ED-24, p.254, 1977.
- [17] P.K. Ko, private communication.
- [18] Y.W. Sing and B. Sudlow, IEDM Tech. Dig., p.732, 1980.
- [19] P.E. Cottrell et al., IEEE Trans. Electron Dev., ED-26, p.520, 1979.
- [20] P.J. Tsang et al., IEEE Trans. Electron Dev., ED-29, p.590, 1982.



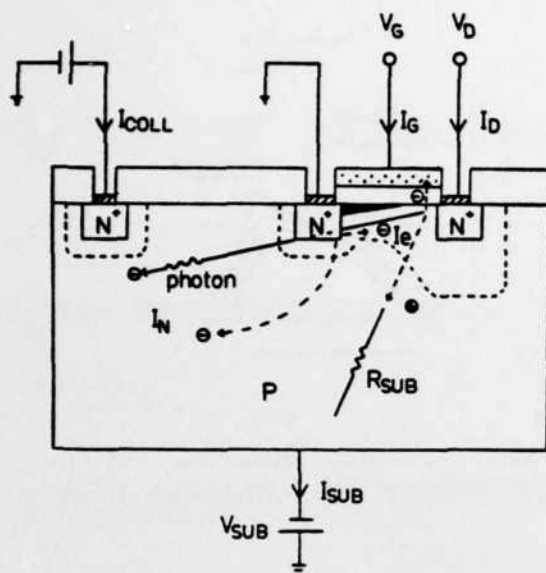


Fig. 1  
Components of hot-electron currents.

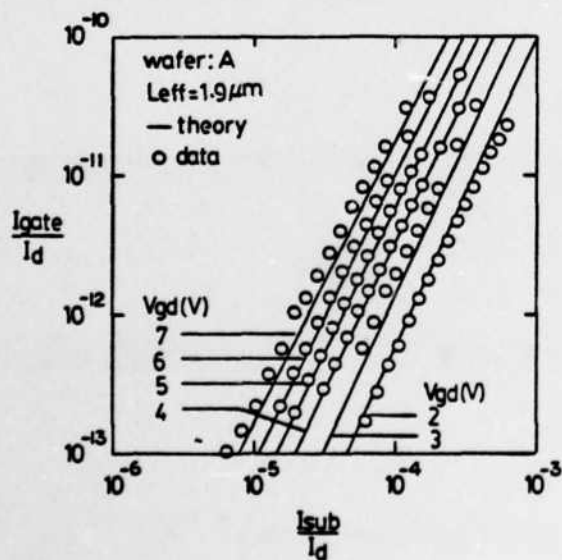


Fig. 3  
The device has 82 nm gate oxide. The solid lines are the theoretical universal correlations between  $I_{sub}$  and  $I_g$ , each applicable for a specific oxide field  $E_{ox} = V_{gd}/x_{ox}$ .

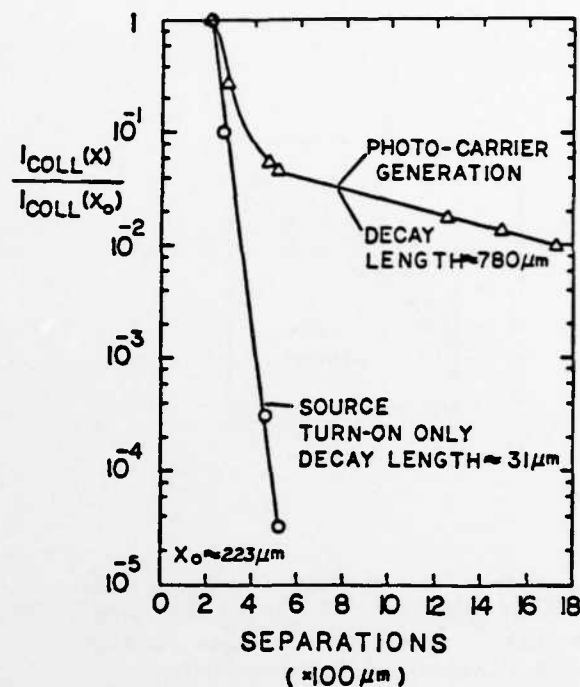


Fig. 2  
Collected substrate electrons (leakage current) vs. the spacing between the collecting node and the MOSFET. Two distinct mechanisms are evident.

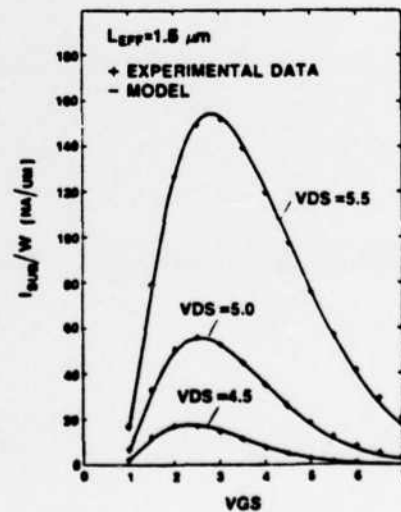


Fig. 4  
The theoretical model is similar to Eq. 7. After Sing and Sudlow [18].

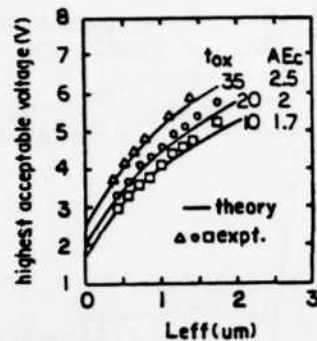


Fig. 5

Theoretical and measured voltages that produce  $I_g = 10^{-15} I_d$ . These are considered maximum acceptable voltages without causing long-term device degradations.

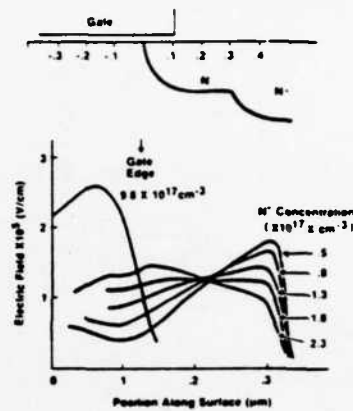


Fig. 6

Lightly-doped-drain structure and the channel field profile. After Tsang et al.[20].

# Hot Electron Induced Photo-carrier Generation in Silicon MOSFETs

*S. Tam and C. Hu*

Department of Electrical Engineering and Computer Sciences  
and the Electronic Research Laboratories,  
University of California, Berkeley, CA 94720.

## ABSTRACT

The phenomenon of and the physical mechanisms for the generation of minority carriers in the substrate of NMOS and CMOS are studied. Secondary impact-ionization is not responsible. The responsible mechanisms are hot-electron induced photo-carrier generation and, under extreme conditions, forward biasing of the source-substrate junction. The photon generation is believed to be due to the bremsstrahlung of the channel hot-electrons. A theoretical model based on the lucky electron concept and bremsstrahlung mechanism is proposed. The calculated characteristics of photon generation agree well with experimental results. About  $2 \times 10^{-5}$  photo-generated minority-carriers are generated for every (primary) impact-ionization event in n-MOSFET. Photo-

- ii -

carrier induced leakage current can be fitted with either a square dependence on distance or an exponential dependence with effective decay length of about  $780\mu m$ .

August 18, 1983

# Hot Electron Induced Photo-carrier Generation in

## Silicon MOSFETs

*S. Tam and C. Hu*

Department of Electrical Engineering and Computer Sciences

and the Electronic Research Laboratories,

University of California, Berkeley, CA 94720.

### 1. Introduction

The presence of and the circuit performance degradation due to minority carriers in MOS IC substrates have been widely recognized [1,2,3]. Excess minority carriers in the substrate are believed to degrade the holding time in dynamic RAMs and to induce errors in digital logic circuits. It has been verified experimentally that the minority carriers in the substrate originate from the peripheral MOSFETs which are operating in the saturation regime. Two mechanisms have been proposed for the generation of the minority carriers in the substrate. They are, (i) the secondary-impact-ionization by holes (in NMOS) that constitute the substrate current [1,2] and (ii) the injection of minority carriers from the source junction caused by the forward biasing due to the flow of the substrate current [3].

It has been shown that secondary-impact-ionization is not justified theoretically [3] while the injection of minority carriers from the source junction into

the substrate is not the only mechanism. A new mechanism which attribute the generation of minority-carriers by photons [4,5] has arecently been proposed. This photo-carrier generation mechanism is fundamentally related to the presence of hot-electrons at the drain end of the scaled MOSFET. Only when the substrate current becomes excessively high, will the ohmic drop in the substrate forward bias the source to substrate junction sufficiently and make the injection of minority carriers possible [6].

Using experimental results from NMOS and CMOS test structures, the two mechanisms of minority carrier generation (photo-carrier generation and source minority-carrier injection) are illustrated. A theoretical model based on the lucky electron concept will be presented to model the hot-electron induced photo-carrier generation mechanism. This model will be subsequently compared to the experimental results. The supression of the injection of minority carriers from the source junction will be demonstrated.

## 2. Experimental Techniques

The presence of minority carriers in the substrate can be detected using the experimental configuration shown in figure(1) on NMOS wafers. A reverse biased pn-junction is shown at a fixed separation from a MOSFET biased in the saturation region. Since this pn-junction is reverse biased, it acts as a collector of minority carriers in the substrate. In the following, we shall call this junction the collecting junction. The above structure is placed inside a light shielded metal enclosure. The current  $I_{COLL}$  through the collecting junction is monitored by an electrometer. The drain to source current  $I_S$  and the substrate current  $I_{SUB}$  of the MOSFET are also measured.

In figure(2), the experimental configuration used on a p-well CMOS wafer is illustrated. The collecting junction ( $p^+n$ -junction) is located outside the p-well and is a collector of holes (minority carrier) in the n-substrate. The well-to-

substrate junction is reverse biased and hence the collector can collect minority carrier (electrons) generated inside the p-well. The reverse bias between the well and the substrate also inhibits the flow of holes from inside the well to the n-substrate. An N-channel MOSFET located inside the p-well is biased in the saturation region and the currents  $I_{HCOLL}$ ,  $I_N$ ,  $I_S$  and  $I_{SUB}$  shown in figure(2) are simultaneously monitored.

### 3. Experimental Results

Typical experimental results of the MOSFET-collector pair for the NMOS wafer are shown in figure(3) and (4). In figure(3), the measured currents are plotted against the gate-to-source voltage ( $V_{GS}$ ) whereas in figure(4), the currents are plotted against the drain-to-source voltage ( $V_{DS}$ ). It is apparent that a correlation between the current collected by the collecting junction and the substrate current exists. This is demonstrated in figure(5) by the log-log plot of the data from figure(4). Referring to figure(5) we can identify two regimes. When the substrate current is not high, the collected minority-carrier current increases almost (but less than) linearly with the substrate current.

When the substrate current is high, the collected minority carrier current increases nearly exponentially with the substrate current (figure(5)). This indicates that when the substrate current is very high, the ohmic drop caused by the flow of the substrate current can forward bias the source-to-substrate junction and cause minority carriers to be injected from the source into the substrate. An important parameter in this situation is the effective substrate resistance  $R_{SUB}$ . Theoretical considerations showed that  $R_{SUB}$  is a function of channel width, channel length and substrate doping level [6]. The near-exponential regime may begin only when  $I_{SUB}R_{SUB} > 0.6V$ . An  $R_{SUB}$  value of approximately  $1k\Omega$  is obtained from the data in figure(5). By incorporating external resistances ( $R_{EXT}$ ) or by applying a slight positive voltage to the substrate, the



injection of minority carriers from the source junction can be enhanced. Figure(6a) and (6b) illustrates the enhancement. Figure(6a) confirms that injection of minority carriers from the source junction begins to occur when the substrate potential reaches about 0.6V above the source potential. Reverse substrate bias can suppress this minority carrier injection mechanism as shown in figure(7). One can conclude that by proper scaling of the substrate doping or using epitaxial substrate to reduce  $R_{S/B}$  and by the use of reverse substrate bias, the injection of minority carriers from the source into the substrate can be significantly or effectively suppressed.

In contrast, the relationship between the collected minority-carrier current and the substrate current in the low substrate current regime is unaffected by the addition of  $R_{EXT}$  or variation in  $V_{S/B}$  as shown in figures(6) & (7). This indicates that the injection of minority carriers from the source is not responsible for the minority carriers in the bulk when  $I_{S/B}$  is moderate or low. It has been suggested that secondary impact-ionization due to holes (ie.  $I_{S/B}$ ) in NMOS introduce the minority carriers in the substrate [1,2]. A rather simplified analysis made by Eitan et al. [3] showed the secondary-impact-ionization to be  $10^9$  times too weak to explain the observed  $I_{COLL}$ . We had independently arrived at the same conclusion following a more rigorous analysis which found the effect to be  $10^{10}$  times too weak.

The correlation between  $I_{S/B}$  and  $I_{COLL}$  is essentially independent of the bias voltages, channel current, channel length, gate oxide thickness, and all other device parameters. The independence of this correlation on  $V_{GS}$ ,  $V_{DS}$ , and  $I_{DS}$  is demonstrated in figure(8). Here, we have included the relationship between  $I_{COLL}$  and  $I_{S/B}$  at three effective gate voltage levels. The gate voltages and drain currents in the three cases vary by the ratio 1:2:3. Yet, we see that the  $I_{COLL} - I_{S/B}$  relationship is essentially unchanged.

The dependence of the collection of minority carriers on the spatial separation between the collecting junction and the MOSFET gives some insight into the transport properties of the minority carriers. In figure(9), two cases of interest are compared. In one case, an external resistance ( $1M\Omega$ ) was used to enhance the injection of minority carriers from the source-to-substrate junction. In another case, the MOSFET was operated in the low substrate current regime ( ie. before the turn-on of the source-to-substrate junction ). The spatial dependence of the two cases are quite different. A simple exponential decay is observed for minority carrier injection from the source where a decay length of about  $31\mu m$  is obtained. Using the diode-reverse-recovery technique, the minority-carrier lifetime was determined to be about  $0.23\mu sec.$ . This decay length is interpreted as the minority carrier diffusion length. The low  $I_{SUB}$  regime curve appears to have two decay lengths. When the separation is small, the decay length is similar to the previous case. However when the separation becomes large, the effective decay length becomes much longer and a value of  $780\mu m$  is arrived. The long decay length part of the curve can also be fitted to the form  $\frac{1}{x^2}e^{-\frac{x}{L}}$  with a decay length of about 1 mm. This difference in the spatial dependence suggests that diffusion of minority carriers cannot explain the transport mechanism in the low substrate current regime. Therefore, neither the injection of minority carriers from the source nor secondary-impact-ionization can explain the experimental results.

A new mechanism has been proposed in which photons emitted from the high-field regions near the drain of the MOSFET generate electron-hole pairs in the substrate [4,5]. Photons with energy more than a few tenths of eV above the silicon band-gap are absorbed very near the MOSFET and the minority carriers generated then spread by diffusion. Photons with energy very close to the band-gap have relatively low absorption coefficient and hence can travel a long

distance before they are absorbed. The combination of the above two transport mechanisms can qualitatively explain our experimental results in figure(9).

Direct observation of light emission from the drain end of the MOSFET confirm this proposal [7]. In figure(10a), we have shown a time-exposure photograph of a MOSFET operating in the saturation regime. A double exposure is taken to illustrate the background structure of the MOSFET. The IV curves of the device is illustrated in figure(10b). The device shown in figure(10a) has a channel width of  $100\mu m$ , channel length of  $1.1\mu m$ , and oxide thickness of  $37nm$ . Under the microscope, a yellowish-white filament of light along the width of the drain can be observed fairly easily even at lower  $V_{DS}$ . The intensity of the light appears to be fairly uniform along the width when  $V_{DS}$  is not very high. At higher  $V_{DS}$ , spots with higher intensities emerge which indicates the presence of localized "hot-spots". The direct observation of light emitted from the drain end of the MOSFET provides good confirmation of the proposed photocarrier generation model.

Experimental results from the p-well CMOS wafer are even more convincing than that of the NMOS wafer. Figure(11) and (12) are typical results plotted against  $V_{GS}$  and  $V_{DS}$  respectively. The presence of  $I_{HCOLL}$  can not be explained by either the secondary-impact-ionization, or source injection since these mechanisms can not give rise to holes outside the well. The holes are generated by photons as illustrated in figure(2). A correlation between the currents  $I_N$  and  $I_{HCOLL}$  with the substrate current  $I_{SUB}$  of the MOSFET is illustrated in figure(13). The relationship between  $I_N$  and  $I_{SUB}$  is almost linear while the relationship between  $I_{HCOLL}$  and  $I_{SUB}$  is slightly sub-linear. In figure(14), the spatial dependence of the  $I_{HCOLL}$  is presented. An effective decay length about  $702\mu m$  is observed. This is in good agreement with the effective decay length obtained from the NMOS wafer.

The current  $I_N$  deserves some explanation. Since the p-well to the n-substrate junction is reverse biased, minority carriers inside the well ( ie. electrons ) and just outside the well ( holes ) will be collected by this junction and give rises to  $I_N$ . The depth of the p-well in this wafer is about  $4\mu m$ . Since most of the photons will be absorbed not too far away from the p-well junction,  $I_N$  therefore can be interpreted as the total minority-carrier generated per second by photons originated from the MOSFET. The presence of  $I_{HCOLL}$  in the n-substrate on the other hand is due to photons with near band-gap energies. If we assume that the near band-gap-photon component is small compared to the total number of photons generated and one photon can only generate one electron-hole pair, then the ratio  $\frac{I_N}{I_{SVB}}$  may be interpreted as the ratio of the rate of photo-carrier generation to the rate of impact-ionization. (The substrate current  $I_{SVB}$  is a measure of the population of hot-electrons in the MOSFET.) This ratio is plotted against the normalized substrate current in figure(15). A ratio of about  $2 \times 10^{-4}$  is obtained which agrees with the value of  $\sim 3 \times 10^{-4}$  obtained by Matsunaga et. al. [2] and  $5 \times 10^{-5}$  obtained by Childs et al. [8] considering the potential inaccuracies of the measurements. This ratio varies only by a factor of 4 when the normalized substrate current varied by about five decades with changing  $V_{DS}$  and  $V_{GS}$  combinations. From our experimental results, it appears that the ratio is about  $4 \times 10^{-5}$  at low  $\frac{I_{SVB}}{I_{DS}}$  and decreases with increasing  $\frac{I_{SVB}}{I_{DS}}$ . Childs et al. [8] however found this probability to be relatively constant with  $V_{DS}$  and  $V_{GS}$ . The cause of the difference between our observation and Child et al. [8] may be that they only measured this ratio for  $\frac{I_{SVB}}{I_{DS}}$  ranging from  $\sim 10^{-5}$  to  $10^{-3}$ .

#### 4. Physical Mechanism and Model

The phenomenon of light emission from avalanching silicon pn-junctions (diodes) has been studied extensively [9-14]. This light emission process was attributed to the radiative interband or intraband transitions. The specific process considered likely were (1) radiative transition of holes between the light-hole band and the heavy-hole band [11], (2) radiative recombination between a hot electron and a free hole [8], and (3) the bremsstrahlung radiation due to the scattering of the hot electrons by charged Coulombic centers [12,13,14].

The experimental results of Matsunaga et al. [2] showed that the generation of minority carriers in the substrate is proportional to the substrate current in both NMOS and PMOS and they found the ratio of minority carrier generation rate to  $\frac{I_{SUB}}{q}$  to be  $3 \times 10^{-6}$  and  $1 \times 10^{-4}$  in NMOS and PMOS respectively. The similarity between NMOS and PMOS quantum efficiencies suggests that the radiative transitions of holes between the light-hole band and the heavy-hole band is not the mechanism since this mechanism would strongly favor PMOS over NMOS due to the much larger hot-hole population and the much higher hole temperature in PMOS than in NMOS.

It is more difficult to choose between the remaining two mechanisms. However, from the experimental results presented in Section(3), we know that  $I_{COLL}$  and hence the photon generation rate is proportional to  $I_{SUB}^\gamma$  where  $\gamma$  is between 0.7 and 0.9 and independent of  $I_{DS}$  (figure(8)). The rate of recombination between electrons and holes would be proportional to the product of electron density ( $\sim I_{DS}$ ) and the hole density ( $\sim I_{SUB}$ ) or to the product of hot-electron density ( $\sim I_{SUB}$ , which is the creation of the hot-electrons) and the hole density ( $\sim I_{SUB}$ ), both in disagreement with data. This strongly disfavors the direct radiative recombination mechanism. Also, the observation of the polarization of lights from avalanche breakdown in pn-junctions [12,14] tends to

support the bremsstrahlung mechanism.

Using the bremsstrahlung radiation as the principle photon-generation mechanism, we can formulate the following model. The lucky electron model first proposed by Shockley [15] to model the hot-electron effects in silicon pn-junctions are found to describe the impact-ionization and the channel hot-electron injection effects in MOSFETs very well [16,17,18]. Using the same formulation, the probability for the channel channel-hot-electrons to attain kinetic energy in excess of  $U$  can be written as

$$P(U) = e^{\left(-\frac{U}{qE_z\lambda}\right)} \quad (1)$$

where  $q$  is the electronic charge,  $U$  is the kinetic energy of the hot electron,  $E_z$  is the channel electric field and  $\lambda$  is the mean-free-path of the channel-hot-electrons mainly due to optical-phonon scattering. The probability  $Q_\nu d\nu$  of the emission of a photon with the energy in the interval  $h\nu$  to  $h(\nu+d\nu)$  due to one electron passing through  $dx$  (thickness) containing  $N_C$  singly-charged Coulombic centers per volume can be written as (appendix(1))

$$Q_\nu d\nu dx = DN_C \frac{d\nu dx}{m^* U h \nu} \quad (2)$$

where  $U$  is the kinetic energy of the electron,  $m^*$  is the electron conductivity effective mass and  $D=2.8 \times 10^6 q^6$  is a numerical constant (appendix(1)) Based on equation(1), the number of hot-electrons passing through the channel with kinetic energy from  $U$  to  $U+dU$  can be written as

$$\begin{aligned} R(U) dU &= \frac{d}{dU} \left( \frac{I_{DS}}{-q} e^{-\frac{U}{qE_z\lambda}} \right) dU \\ &= \frac{I_{DS}}{q^2 E_z \lambda} e^{-\frac{U}{qE_z\lambda}} dU \end{aligned} \quad (3)$$

where  $I_{DS}$  is the channel current of the MOSFET. The total number of photon generated at the frequency interval from  $\nu$  to  $\nu+d\nu$  due to the hot-electrons having the kinetic energy from  $U$  to  $U+dU$  is

$$\begin{aligned}
 & \int_0^{L_{EFF}} Q_\nu d\nu R(U) dU dx \\
 &= \int_0^{L_{EFF}} \frac{D d\nu N_C I_{DS} e^{-\frac{U}{qE_x \lambda}}}{m^* U h \nu q^2 E_x \lambda} dU dx.
 \end{aligned} \quad (4)$$

From equation(4), we can evaluate the total energy radiated in the frequency range  $\nu$  to  $\nu+d\nu$  per unit time due to all hot-electrons with initial kinetic energy above  $h\nu$  as

$$\begin{aligned}
 W_\nu d\nu &= h\nu d\nu \int_0^{L_{EFF}} \int_{h\nu} \frac{D d\nu N_C I_{DS}}{m^* q^2 E_x \lambda} \frac{e^{-\frac{U}{qE_x \lambda}}}{U} dU dx \\
 &= \frac{D d\nu N_C I_{DS}}{m^* q^2 \lambda} \int_0^{L_{EFF}} \frac{E_1\left(\frac{h\nu}{qE_x \lambda}\right)}{E_x} dx \\
 &\approx \frac{D d\nu N_C I_{DS}}{m^* q h \nu} \int_0^{L_{EFF}} e^{-\frac{h\nu}{qE_x \lambda}} dx
 \end{aligned} \quad (5)$$

where  $E_1$  is the exponential integral. In obtaining equation(5a), we have assumed that the maximum channel electric field is a few times  $10^5 \text{ Vcm}^{-1}$  and the hot-electron scattering mean-free-path is on the order of  $10 \text{ nm}$  [17,18]. Therefore, the product  $qE_x \lambda$  is at most a few times  $0.1 \text{ eV}$ . Since we are only interested in photons with energy greater than  $1.12 \text{ eV}$ , the argument in  $E_1$  in equation(5) is usually greater than 5 and we have  $E_1(z) \approx \frac{e^{-z}}{z}$  [20]. By a change of variable and assuming that  $\frac{dE_x}{dx} \approx \text{constant}$ , we have

$$W_\nu d\nu = \frac{D d\nu N_C I_{DS}}{m^* q h \nu} \int_0^{L_{EFF}} \frac{q \lambda}{h \nu \frac{d}{dx} \left( \frac{1}{E_x} \right)} d \left( e^{-\frac{h\nu}{qE_x \lambda}} \right) \quad (6)$$

$$= \frac{D d\nu N_C I_{DS}}{m^* q} \frac{q \lambda E_m^2}{h^2 \nu^2 \left| \frac{dE_x}{dx} \right|_L} e^{-\frac{h\nu}{qE_m \lambda}} \quad (6a)$$

where  $E_m$  is the maximum channel electric field (ie. at the drain end). To obtain equation(6a), we have assumed  $\left| \frac{dE_x}{dx} \right|_L$  is constant and hence can be taken out of the integral. The quantity  $\frac{q \lambda E_m^2}{h \nu \left| \frac{dE_x}{dx} \right|_L}$  has a dimension of length and can



be interpreted as the length of the channel near the drain where the hot-electrons possess enough kinetic energy to induce bremsstrahlung. In figure(16), we have plotted the theoretical photon spectrum (ie.  $\frac{W_\nu}{h\nu}$ ) based on equation(6a) and compared it with that measured by Chynoweth et al. [8]. Reasonable agreement is achieved when  $qE_m\lambda$  is chosen to be 0.35 eV. Using a hot-electron mean-free-path  $\lambda$  of 9.2 nm [18], the field is calculated to be about  $4 \times 10^5 \text{ Vcm}^{-1}$  which is of reasonable magnitude since this is the approximate field in a pn-junction at avalanche breakdown [25].

The number of photons generated per unit frequency per unit time,  $I_\nu$  is  $\frac{W_\nu}{h\nu}$ . In order for the photons to generate minority-carriers in the substrate, the energy of the photons must be greater than  $E_g$  where  $E_g = 1.12 \text{ eV}$  is the bandgap energy of silicon. Denoting  $\frac{E_g}{h}$  as  $\nu_0$ , we have the number of photons in the frequency interval  $\nu_0$  to  $\nu_0 + \Delta\nu$  (ie. in the energy range  $E_g$  to  $E_g + \Delta E$ ) to be

$$I_{\nu_0, \nu_0 + \Delta\nu} = \frac{DN_C I_{DS}}{q m^*} \frac{q \lambda E_m^2}{h^3 \left| \frac{dE_x}{dx} \right|_L} \int_{\nu_0}^{\nu_0 + \Delta\nu} e^{\frac{-h\nu}{qE_m\lambda}} \frac{d\nu}{\nu^3} \quad (7)$$

The exact solution to equation(7) is shown in appendix(2). The total number of photons can be evaluated simply by integrating equation(7) to  $\infty$ . The exact solution is also shown in appendix(2). To a very good approximation, the exact solution shown in appendix(2) can be expressed as,

$$\begin{aligned} I_{\nu_0, \infty} &= \frac{D}{h^2 q} \left( \frac{N_C I_{DS}}{m^*} \right) \frac{q \lambda E_m^2}{E_g \left| \frac{dE_x}{dx} \right|_L} \frac{a}{\nu_0} e^{\frac{-(E_g + b)}{qE_m\lambda}} \\ &= 8.7 \times 10^{-22} (\text{J} \cdot \text{s} \cdot \text{C}^{-1}) \left( \frac{N_C I_{DS}}{m^*} \right) \frac{q \lambda E_m^2}{E_g \left| \frac{dE_x}{dx} \right|_L} \frac{a}{\nu_0} e^{\frac{-(E_g + b)}{qE_m\lambda}} \end{aligned} \quad (8)$$

where  $a = 0.3$ , and  $b = 0.16 \text{ eV}$ .

In the MOSFET, the impact-ionization substrate current can be formulated as [17,18,20]

$$I_{SUB} = \frac{I_{DS} A_i E_m^2}{B_i \left[ \frac{dE_x}{dx} \right] E_m} e^{-\frac{B_i}{E_m}} \quad (9)$$

where  $A_i$  and  $B_i$  are the pre-exponential and exponential constants in the ionization coefficient respectively [25]. By eliminating the maximum channel field  $E_m$  between equation(10) and equation(12), we obtain

$$I_{\nu_{ph}} = a \cdot 6.7 \times \frac{10^{-22} N_C I_{DS}}{\nu_{ph} m^*} \frac{q \lambda E_m^2}{E_g \left[ \frac{dE_x}{dx} \right]_L} \left\{ \frac{I_{SUB} B_i \left[ \frac{dE_x}{dx} \right] E_m}{I_{DS} A_i E_m^2} \right\}^{\frac{(\hbar \nu_{ph} + b)}{q \lambda B_i}} (s^{-1}) \quad (10)$$

Since  $E_g + b = 1.28 eV$  and  $q B_i \lambda \approx 1.24 eV$  [18], we have  $\frac{(E_g + b)}{q B_i \lambda} \approx 1$ . Then equation(13) is reduced to

$$I_{\nu_{ph}} = 7.4 \times 10^{-37} \left[ \frac{N_C q B_i \lambda I_{SUB}}{m^* A_i} \right] (s^{-1}) \quad (11)$$

The total minority-carrier current generated is

$$q I_{\nu_{ph}} = 7.4 \times 10^{-37} q \left[ \frac{N_C q B_i \lambda I_{SUB}}{E_g m^* A_i} \right] \quad (12)$$

$$= 1.8 \times 10^{-5} I_{SUB} \quad (\text{for NMOS}) \quad (12a)$$

It is seen from equation(11) that the total photon generation rate is proportional to the substrate current  $I_{SUB}$  and independent of the channel current  $I_{DS}$  in agreement with the data presented in Section(3). In obtaining equation(12a), we have used  $q B_i \lambda = 1.24 eV$  [18],  $E_g = 1.12 eV$ ,  $A_i = 1.6 \times 10^7 m^{-1}$  [18]  $m^* = 0.26 m_0 = 0.26 \times 9.11 \times 10^{-31} kg$  [26], and  $N_C$  equals to the drain doping concentration  $\sim 5 \times 10^{20} cm^{-3} = 5 \times 10^{26} m^{-3}$ . The ratio,  $\frac{q I_{\nu_{ph}}}{I_{SUB}}$ , of  $1.8 \times 10^{-5}$  obtained is in excellent agreement with our experimental results and those reported in the literature [2,8,27]. The values of  $A_i$  and  $B_i$  are less accurately known for PMOS

(holes). If one uses  $qE_1\lambda = 1.47eV$ ,  $m^* = 0.386m_0$ , and  $A_1 = 2.21 \times 10^6 m^{-1}$ , the ratio is found to be  $1 \times 10^{-4}$ . There is some evidence that  $A_1$  is proportional to  $E_m$  (ie.  $\frac{I_{SUB}}{I_{DS}}$ ) [17,25]. That and equation(12) may explain the decrease of the ratio at high  $I_{SUB}$  such as evident in figure(15).

In the above estimation, we have assumed that the  $N_C$  is equal to the drain doping density. That is, we have assumed that the ionized donors at the drain are causing the scattering of the hot-electron. This is similiar mobility degradation phenomenon found in heavily doped semiconductors.

The photons that can travel a long distance to generate minority carriers are those that have energy very close to  $E_g = h\nu_0$ . From equation(7a), the rate of generation of photons with energy between  $\nu_0$  and  $\nu_0 + \Delta\nu$  is (ie.  $\frac{W_{\nu_0}}{h\nu_0} \Delta\nu$ )

$$I_{\nu_0, \nu_0 + \Delta\nu} \sim e^{\frac{-E_g}{qE_m\lambda}} \Delta\nu \quad (1)$$

Eliminating  $E_m$  from equation(15) using equation(12) and one obtains

$$I_{\nu_0, \nu_0 + \Delta\nu} \sim \left[ I_{SUB} \right]^{\frac{E_g}{qB_1\lambda}} \quad (1)$$

$$\sim \left[ I_{SUB} \right]^{0.65}$$

This is in excellent agreement with the slopes in figure(5), figure(6), and figure(13) ( $I_{HCOLL}$ ). Equations(1 ) and (1 ) can explain the two different slopes in figure(13). In figure(18), we have plotted  $I_{\nu_0, \nu_0 + \Delta\nu}$  as a function of  $I_{SUB}$  using  $\frac{\Delta\nu}{\nu_0}$  as the parameter (equation(1 ) and (1 )). When the ratio  $\frac{\Delta\nu}{\nu_0}$  decreases, the relationship between the photon generation rate (in the range  $\nu_0$  to  $\nu_0 + \Delta\nu$ ) and  $I_{SUB}$  is sub-linear. This also agrees with our experimental results shown in Section(3) (eg. figure(13)).

In order to calculate the number of minority-carriers generated per unit

time as a function of location in the wafer, we have to solve the three dimensional current continuity equation subjected to the photon generation rate described above, and the geometries of the MOSFET and the collector pair, and also we have to know the accurate dependence of the absorption coefficient on photon energy. Due to the complexity of this part of the problem, this will not be treated here.

## 5. Conclusion

The phenomenon of minority-carrier generation in the substrate of VLSI chips is studied. Based on experimental results on NMOS and CMOS wafers, we concluded that the minority-carrier generation mechanisms are (i) injection of minority-carriers from the source to substrate junction when the substrate current is excessively high, and (ii) photo-carrier generation where the photons originate from the drain high-field region of the MOSFET. Specifically, secondary-impact-ionization is found to play no part in the creation of substrate minority-carriers.

New evidence rules out the hot-hole interband transitions and tend to contradict the electron-hole recombination as the mechanism of light generation. No contradiction is found with the bremsstrahlung explanation. A theoretical formulation of the hot-electron induced photon generation phenomenon is presented. The theory is based on the bremsstrahlung of the hot-electrons. Using this approach and the lucky electron concept, the theoretical quantum efficiency of photon generation is obtained. Assuming that one photon with energy above the Si band-gap can generate only one electron-hole pair, we obtained the theoretical minority-carrier generation rate. A theoretical value for the minority-carrier generation rate of  $1.8 \times 10^{-5}$  for each impact-ionization event in NMOS and for PMOS are in good agreement with the experimentally obtained value of  $\approx 2 \times 10^{-5}$  and  $1 \times 10^{-4}$ . The required density of Coulombic

scattering centers was found to be the ionized drain impurity dopant. The generation rate of near bandgap photons increases with  $I_{SUB}$  sublinearly as  $\sim (I_{SUB})^{0.65}$  for NMOS.

The presence of hot-carriers in N-channel MOSFETs is widely recognized [28,29,30]. The observation of the substrate current and the gate current are two examples of hot-electron effects in MOSFETs. Therefore, we should expect the light emission process, which is just another hot-carrier effect, to be present in MOSFETs. It should be noted that the light-emission process is not a direct consequence of impact-ionization, rather, it is merely due to the presence of hot-carriers. Since hot-carriers will also cause impact-ionization and leads to substrate current, we would expect a correlation between the substrate current and the photon-generated minority-carrier current to exist.

Previous studies on light emission from silicon pn-junctions under avalanche breakdown showed that the spectrum of radiation may be approximated as [12]  $W_\nu \sim e^{\frac{-h\nu}{kT_e}}$  where  $T_e$  is the electron temperature of the hot-electrons and  $k$  is the Boltzman's constant. To apply this to the MOSFET, we need to know the relationship between the electron temperature  $T_e$  and the channel electric field as well as the electric field (this relationship is unfortunately not accurately known). Using the lucky electron concept, we have derived  $W_\nu \sim e^{\frac{-h\nu}{qE_m\lambda}}$ , where  $\lambda$  is the hot-electron mean-free-path and  $E_m$  is the peak channel electric field. In essence, we have shown that  $T_e = \frac{q}{k}E_m\lambda$  [31]. Furthermore, a correlation between the photon generation rate, and hence the minority-carrier generation rate, and the substrate current is theoretically obtained as stated above.

Form this model, we concluded that a possible way to reduce the photo-carrier ge neration is the use of lightly-doped source-drain structure [32] or

other improved structures in the design of VLSI MOSFETs. On the other hand, the common technique of employing guard-rings or dummy collectors is less effective in combating the long-range substrate leakage current described here as may be expected. The photo-carrier induced leakage current decays with distance roughly according to the diffusion length at close range. At long range, it can be fitted to either a square dependence on distance or an exponential dependence with an effective decay length of about  $780\mu m$ .

## 6. Acknowledgement

The many valuable discussion with Dr.P.Ko and Dr.F.C.Hsu concerning the subject is greatly appreciated. The authors are greatly indebted to their critical comments. The interest in the subject and the valuable comments from S.Wong and P.Li is also appreciated. Research sponsored in part by DARPA under grant N00039-81-K-0251 and AFOSR under grant .

## Appendix(1) : Formulation of the Basic Equations

We shall in this appendix, formulate the basic equations used in the model of bremsstrahlung that is presented in section(4) (ie. equations(3) and (4)).

Based on the classical electromagnetic theory, when an electron collides with a singly charged Coulombic center, the electron will move on a hyperbolic orbit subjected to pure Coulombic interaction and will give off energy according to the following equation [33]

$$\frac{q^2 \epsilon_{Si}}{6\pi \epsilon_0 c^3} \left( \frac{q^2}{4\pi \epsilon_{Si}^* \epsilon_0 r^2} \right)^2 \quad (A1.1)$$

where  $c$  is the speed of light in vacuum,  $\epsilon_0$  is the permittivity of free space,  $\epsilon_{Si}$  is the relative dielectric constant of Si,  $\epsilon_{Si}^*$  is the effective dielectric constant in relation to the hot-electrons,  $q$  is the electronic charge,  $c$  is the speed of light, and  $r$  is the distance between the electron and the Coulombic center. Kramer

[21], starting with the rate expression, formulated a theory to explain the phenomenon of continuous X-ray spectrum which is commonly known as the bremsstrahlung radiation.

By following Kramer's analysis, we obtained for this problem

$$(A1.2) \quad \int \nu d\nu dx = \frac{32\pi^2 \epsilon_0^2 q^2 d\nu N_C dx}{3 c^2 \epsilon_0^2 (4\pi\epsilon_0)^3 U}$$

where  $\nu$  is the energy radiated for the entire passing per unit frequency ( $\nu$ ) for one electron passing through  $dx$  (thickness),  $N_C$  is the volume density of the Coulombic centers, and  $U$  is the initial kinetic energy of the electron. Equation(A1.2) is obtained by averaging over all impact parameter (distance between the Coulombic center and the projection of the electron orbit at infinity) of the electron collision. It was pointed out by Figielski et al. [11], that the effective dielectric constant used should be very close to unity. In this paper, we have therefore assumed  $\epsilon^*_{\nu} = 1$ . Implicit in equation(A1.2) is the quantity  $Q_{\nu} = \frac{h\nu}{\epsilon^*_{\nu}}$ , which has the meaning of the probability that a photon at energy  $h\nu$  is emitted due to a single electron collision with the Coulombic centers. By inserting equation(A1.2) to the meaning of  $Q_{\nu}$ , we obtain equation(2).

### Appendix(2) : Integrating Equation(8)

The exact solution to the integral in equation(8) is

$$(A2.1) \quad I_{\nu, \nu_0 + \Delta\nu} = \frac{DN_C I_{DS}^L}{q m^2 h^2} \times \left[ \frac{e^{-\frac{h\nu_0}{q E_m \lambda}}}{1 + \frac{\nu_0}{\Delta\nu}} + \left[ E_1 \left( \frac{q E_m \lambda}{h(\nu_0 + \Delta\nu)} \right) - E_1 \left( \frac{q E_m \lambda}{h\nu_0} \right) \right] \right]$$

need to revise

and

$$(A2.2) \quad I_{\nu_0} = \frac{DN_C I_{DS}^L}{q h m^2 h^2} \left[ \frac{e^{-\frac{h\nu_0}{q E_m \lambda}}}{\frac{q E_m \lambda}{h\nu_0}} - \frac{E_1 \left( \frac{q E_m \lambda}{h\nu_0} \right)}{\frac{q E_m \lambda}{h\nu_0}} \right]$$



$$= 8.8 \times 10^{-22} (J \cdot s \cdot C^{-1}) \times \frac{N_C I_S}{Am} \left\{ e^{\frac{-h\nu_0}{qE_m\lambda}} - \frac{h}{qE_m\lambda} E_1\left(\frac{h\nu_0}{qE_m\lambda}\right) \right\} \frac{e^{\frac{-h\nu_0}{qE_m\lambda}}}{\nu_0} \left[ 1 - G\left(\frac{h\nu_0}{qE_m\lambda}\right) \right] \quad (A2.3)$$

where  $G(\zeta)$  is a rational polynomial [29]. In figure(A2.1), we have plotted the function  $1 - G\left(\frac{h\nu_0}{qE_m\lambda}\right)$  versus  $\frac{1}{qE_m\lambda}$ . This can be approximated by  $\alpha e^{-\frac{b}{qE_m\lambda}}$  where  $\alpha = 0.30206$  and  $b = 0.1585 eV$  (dotted curve). In figure(A2.2), we have plotted the quantity inside the {} in equation(A2.1) versus  $\frac{\Delta\nu}{\nu_0}$ . A very good fit to this

quantity is  $1 - e^{-k\frac{\Delta\nu}{\nu_0}}$  where  $k = \frac{1.5 eV}{qE_m\lambda}$ . Hence the quantity  $I_{\nu_0, \nu_0 + \Delta\nu}$  can be written as

$$I_{\nu_0, \nu_0 + \Delta\nu} = I_{\nu_0} (1 - e^{-k\frac{\Delta\nu}{\nu_0}}) \quad (A2.4)$$

## References

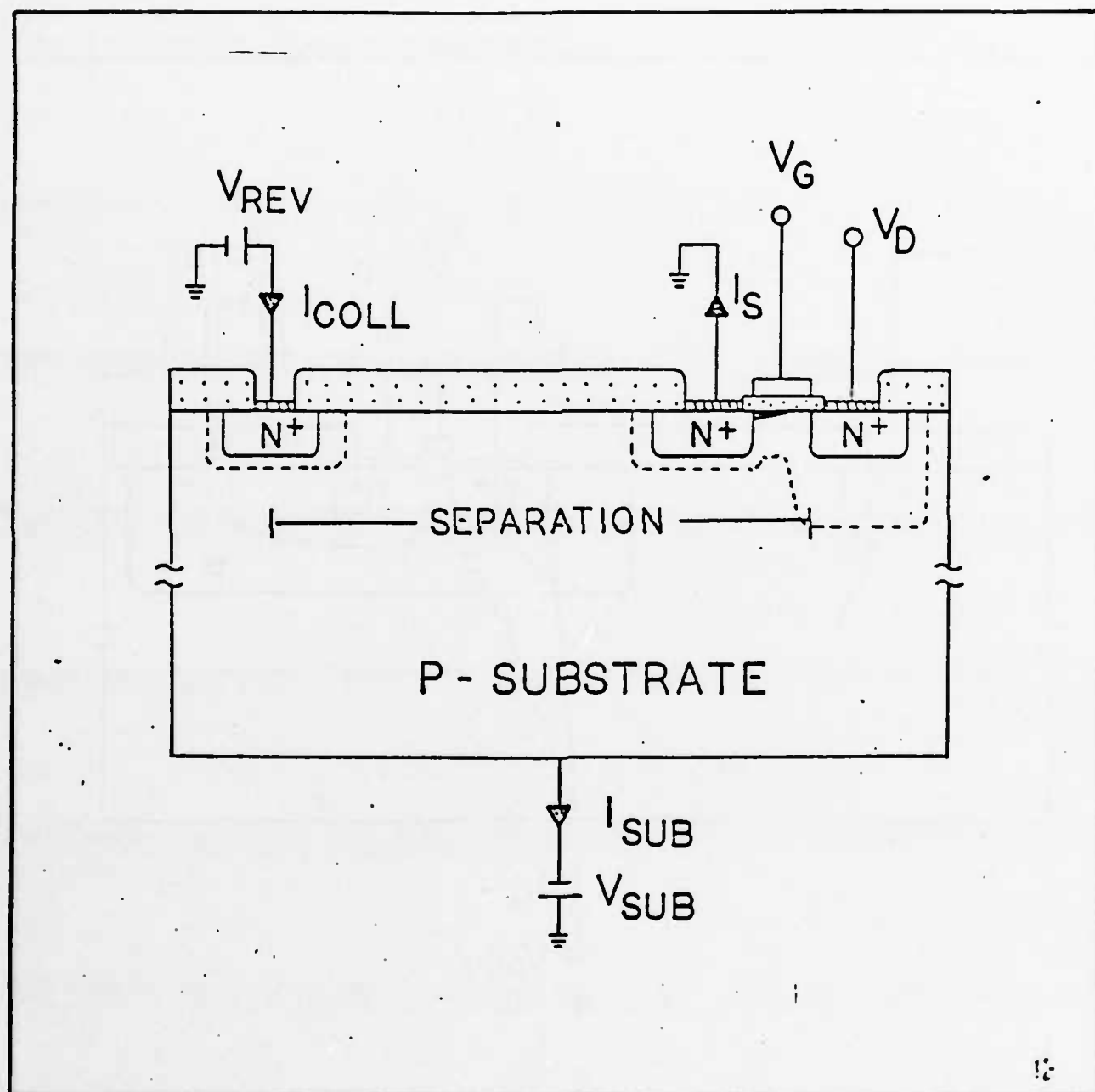
- [1] P.K.Chatterjee, "VLSI dynamic nMOS design constraints due to drain induced primary and secondary impact ionization," presented at the Int. Electron Device Meeting, Washington, DC, Dec. 1979.
- [2] J.Matsunageha, H.Momose, H.Iizuka, and S.Kohyama, "Characterization of two step impact ionization and its influence in NMOS and PMOS's VLSI's," presented at the Int. Electron Device Meeting, Washington, DC., Dec. 1980.
- [3] B.Eitan, D.Frohman-Bentchkowsky, and J.Shappir, "Holding time degradation in dynamic MOS RAM by injection-induced electron currents," IEEE Trans. Electron Devices, vol.ED-28, p.1515, 1981.
- [4] P.A.Childs, W.Eccleston, and R.A.Stuart, "Alternative mechanism for substrate minority carrier injection in MOS devices operating in low level avalanche," Electronics Letters, vol.17, pp.281-282, 1981.
- [5] S.Tam, F.-C.Hsu, P.K.Ko, C.Hu, and R.S.Muller, "Hot-electron induced excess carriers in MOSFET's," IEEE Electron Device Letters, vol.EDL-3, pp.376-378, Dec. 1982.
- [6] F.-C.Hsu, P.K.Ko, S.Tam, C.Hu, and R.S.Muller, "An analytical breakdown model for short-channel MOSFETs," IEEE Trans. Electron Devices, vol.ED-29, pp.1735-1740, Nov. 1982.
- [7] S.Tam, F.C.Hsu, P.K.Ko, C.Hu, and R.S.Muller, "Direct observation of visible-light emission from Si MOSFETs," to be published.
- [8] P.A.Childs, R.A.Stuart, and W.Eccleston, "Evidence of optical generation of minority carriers from saturated MOS transistors," Solid-State Electronics, vol.26, p.685, 1983.

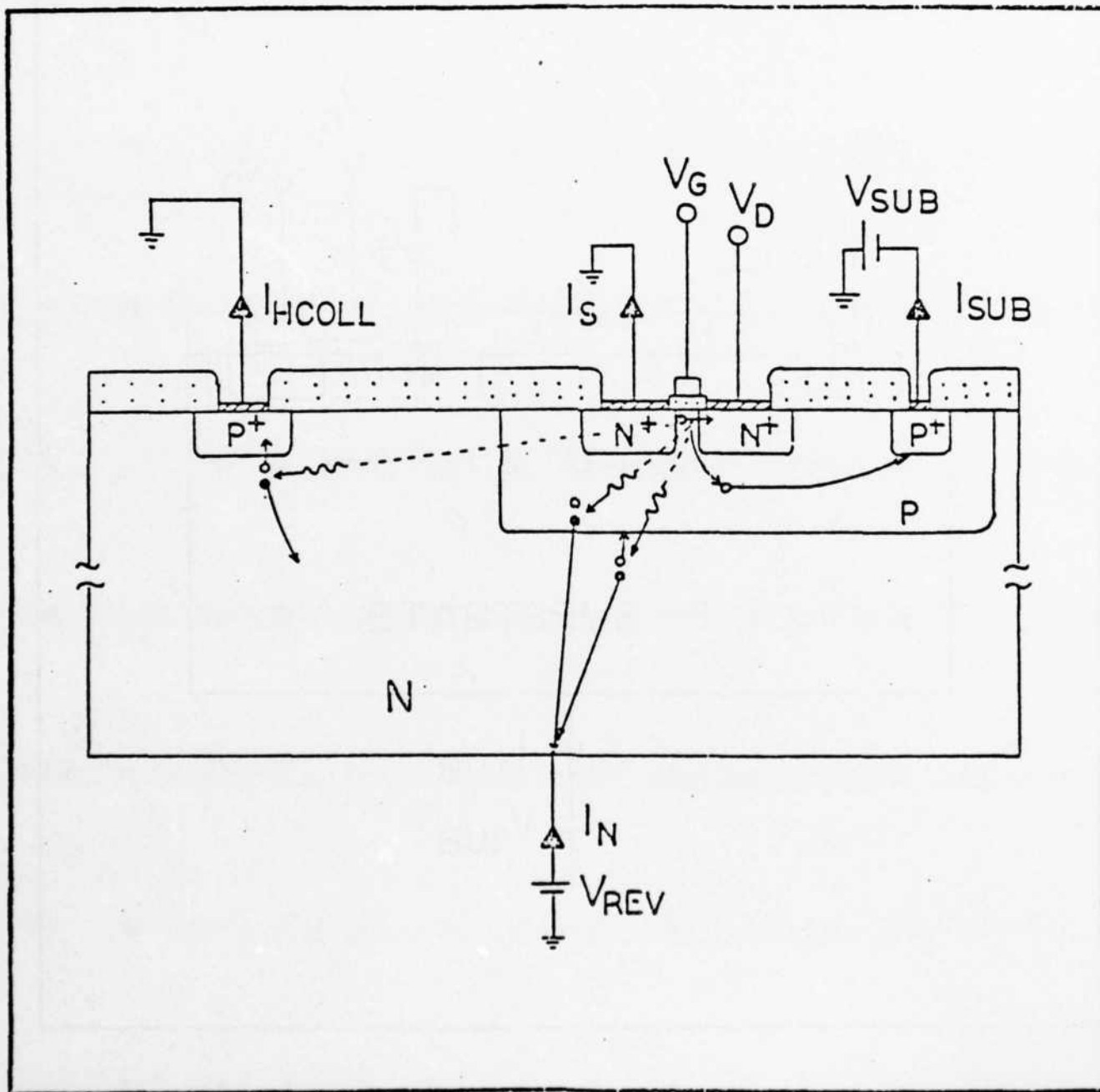
- [9] A.G.Chynoweth and K.G.McKay, "Photon emission from avalanche breakdown in silicon," *Physical Review*, vol.102,no.2, p.369, 1956.
- [10] P.A.Wolf, "Theory of optical radiation from breakdown avalanches in germanium," *J. Phys. Chem. Solids*, vol.16, pp.184-190, 1960.
- [11] W.Haecker, "Infrared radiation from breakdown plasmas in Si, GaSb, and Ge: evidence for direct free hole radiation," *Phys. Stat. Sol. (a)*, vol.25, p.301, 1974.
- [12] T.Figielski and A.Torun, "On the origin of light emitted from reverse biased p-n junctions," *Int. Conf. Phys. Semiconductors, Exeter*, p.863, 1962.
- [13] J.Schwchun and L.Y.Wei, "Mechanism for reverse-biased breakdown radiation in p-n junctions," *Solid-State Electronics*, vol.8, pp.485-493, 1964.
- [14] E.Karnieniecki, "Hot carriers in microplasmas and their radiation in germanium and silicon," *Phys. Stat. Sol.*, vol.6, p.887, 1964.
- [15] W.S. hockley, "Problems related to p-n junctions in silicon," *Solid-State Electronics*, vol2, pp.35-67, 1961.
- [16] C.Hu, "Lucky-electron model of hot electron emission," presented at the *Int. Electron Device Meeting, Washington, DC.*, 1979.
- [17] S.Tam P.Ko, C.Hu, and R.S.Muller, "Correlation between substrate and gate currents in MOSFETs," *IEEE Trans. Electron Devices*, vol. ED-29, pp.29, pp.1740-1744, Nov. 1982.
- [18] S.Tam et. al. , "Lucky-Electron model of channel hot electron injection in MOSFETs," to be published.
- [19] H.A.Kramers, "On the theory of X-Ray absorption and of the continuous X-Ray spectrum," *Phil. Mag*, vol.46, p.836, 1923.
- [20] P.Ko, R.S.Muller, and C.Hu, "A unified model for hot-electron currents in MOSFET's," in *IEDM Technical Digest*, p.600, 1981.

- [21] M.Abramowitz and I.A.Stefun, editors. Handbook of Mathematical Functions, New York, 1972.
- [22] T.N.Nguyen and J.D.Plummer, "Physical mechanisms responsible for short channel effects in MOS devices," in IEDM Technical Digest, pp.596-599, 1981.
- [23] T.Toyabe, K.Yamaguchi, S.Asai, M.Mock, "A numerical model of avalanche breakdown in MOSFETs," IEEE Trans. Electron Devices, vol.ED-25, pp.825-832, 1978.
- [24] S.Selberherr, A.Shutz, and H.W.Potzl, "MINIMOS - a two-dimensional MOS transistor analyzer," IEEE Trans. Electron Devices, vol.ED-27, pp.1540-1550, 1980.
- [25] S.M.Sze, Physics of Semiconductor Devices, J.Wiley and Sons, 1969.
- [26] R.S.Muller and T.I.Kamins, Device Electronics for integrated circuits, J. Wiley and Sons, 1979.
- [27] R.H.Haitz, "Studies on optical coupling between silicon p-n junctions," Solid-State Electronics, vol.8, p.417, 1965.
- [28] H.N.Yu, A.Reisman, C.M.Osburn, and D.L.Critchlow, "1 $\mu$ m MOSFET VLSI technology: Part 1 - an overview," IEEE J. of Solid-State Circuits, vol.SC-14, pp.240-246, 1979.
- [29] T.H.Ning, P.W.Cook, R.H.Dennard, C.M.Osburn, S.E.Schuster, and H.N.Yu, "1 $\mu$ m MOSFET VLSI technology: Part 5 - hot-electron design constraints," IEEE Trans. Electron Devices, ED-26, p.346, 1979.
- [30] E.Takeda, H.Kume, T.Toyabe, and A.Asai, " Submicrometer MOSFET structure for minimizing hot-carrier generation," IEEE Trans. Electron Devices, vol.ED-29, pp.611-618, 1982.
- [31] S.Tam, F.-C.Hsu, C.Hu, R.S.Muller, and P.K.Ko, "Hot-electron currents in very short channel MOSFETs," IEEE Electron Device Letters, vol.EDL-4,

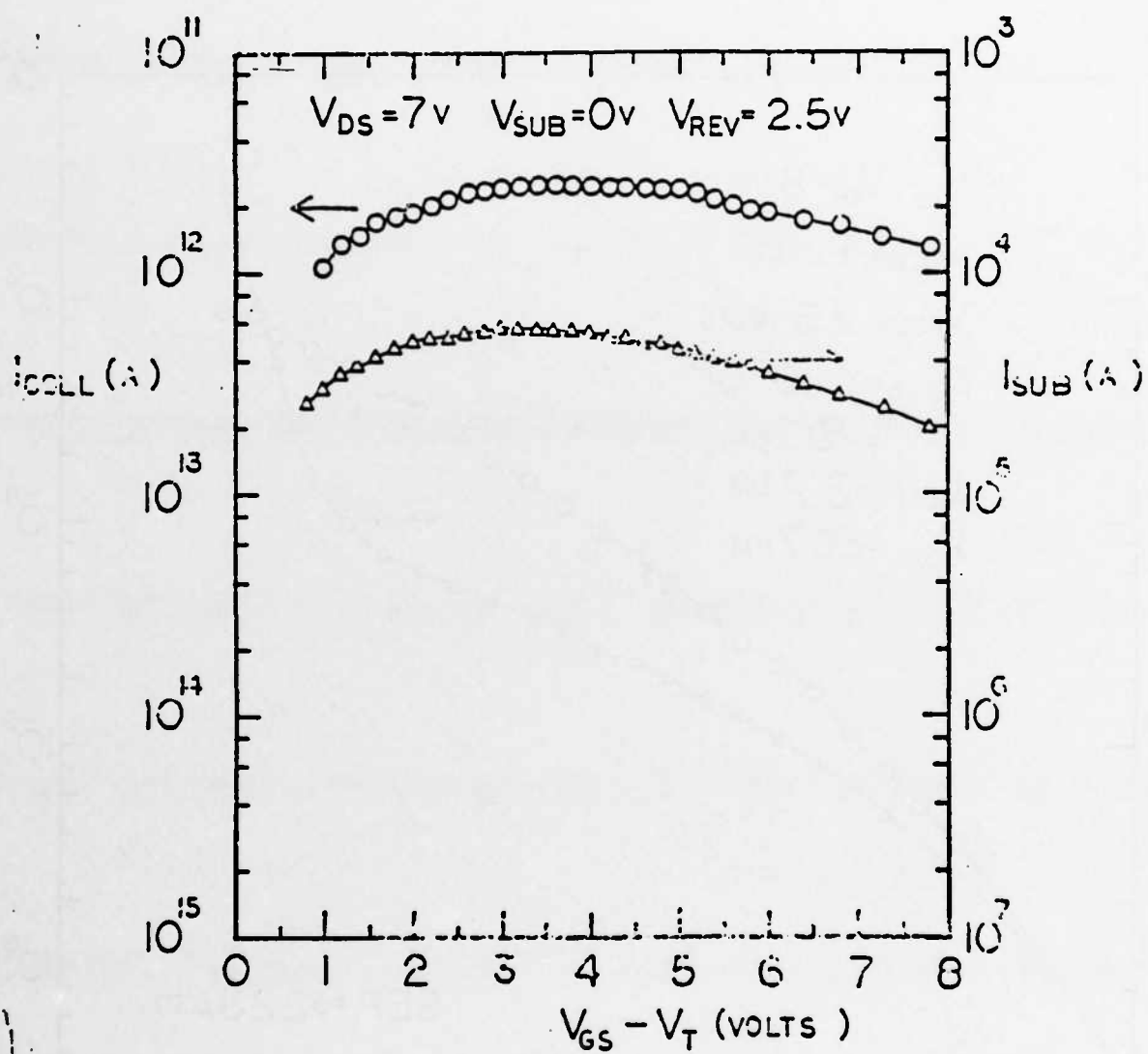
p.249, 1983.

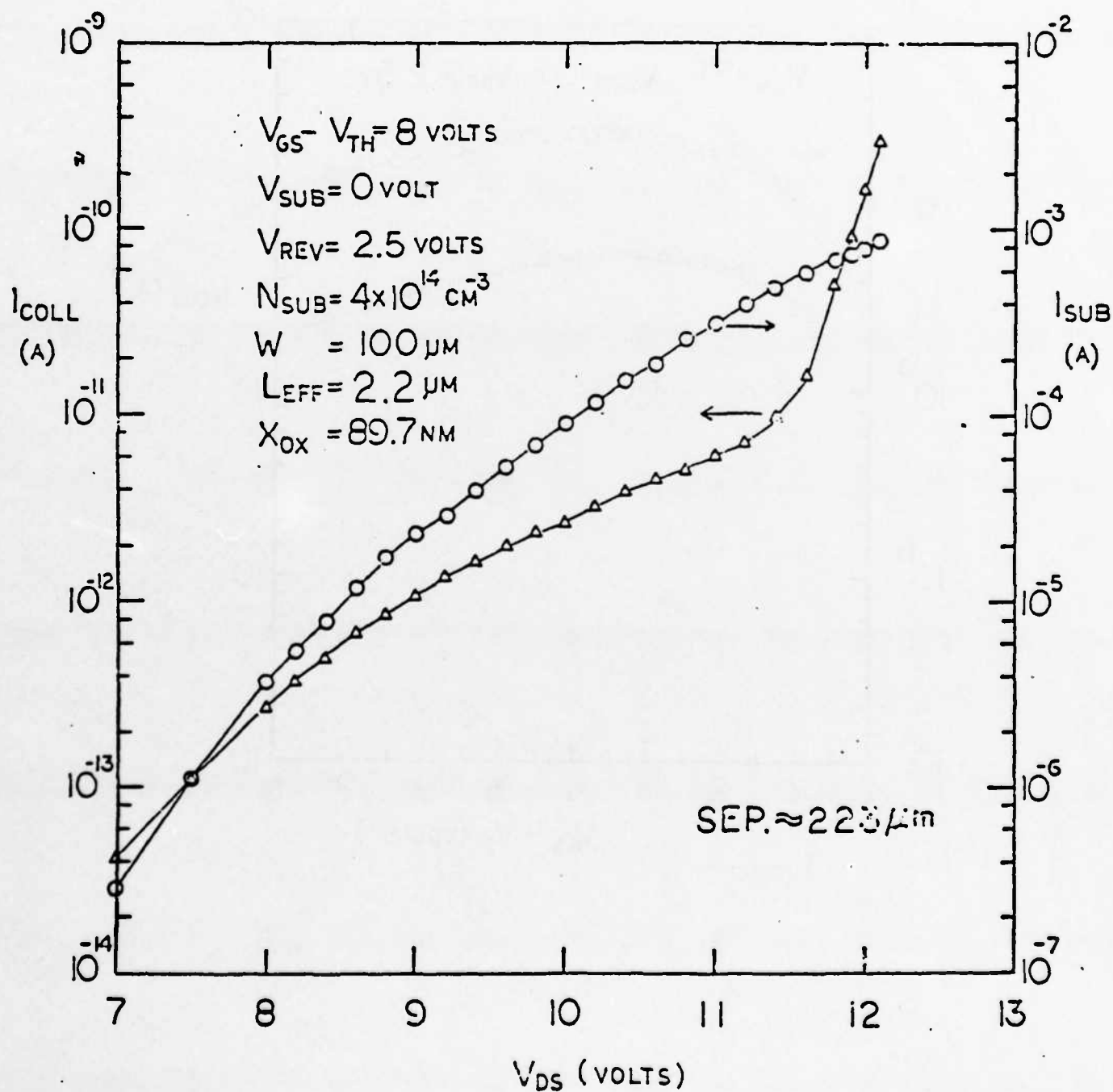
- [32] S.Ogura, P.J.Tsang, W.Walker, D.L.Critchlow, and J.F.Shepard, "Design and characteristics of the lightly doped drain-source (LDD) insulated gate field-effect transistor," IEEE Trans. Electron Devices, vol.ED-27, p.1359, 1980.
- [33] W.K.H.Panofsky and M.Philips, Classical Electricity and Magnetism, Addison-Wesley, 1964.











$I_{\text{COLL}} \text{ (A)}$

$10^{-9}$

$10^{-10}$

$10^{-11}$

$10^{-12}$

$10^{-13}$

$10^{-14}$

SEPARATION =  $223 \mu\text{m}$

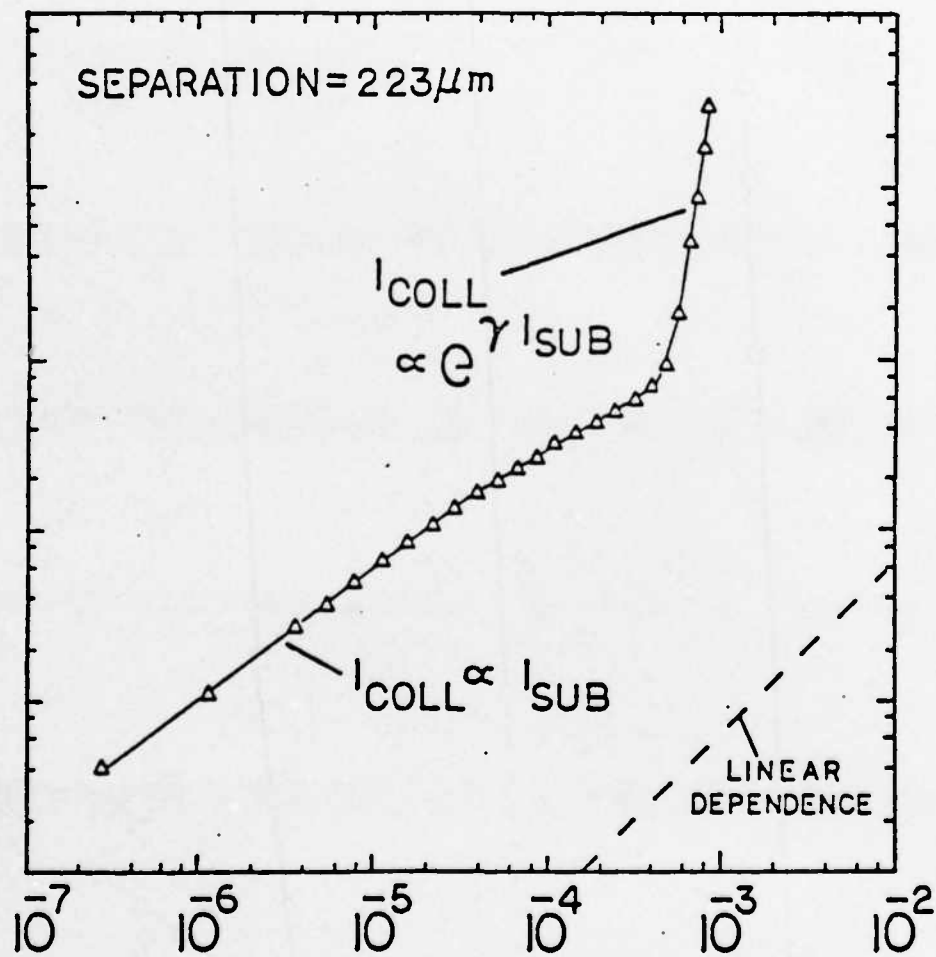
$I_{\text{COLL}} \propto e^{\gamma I_{\text{SUB}}}$

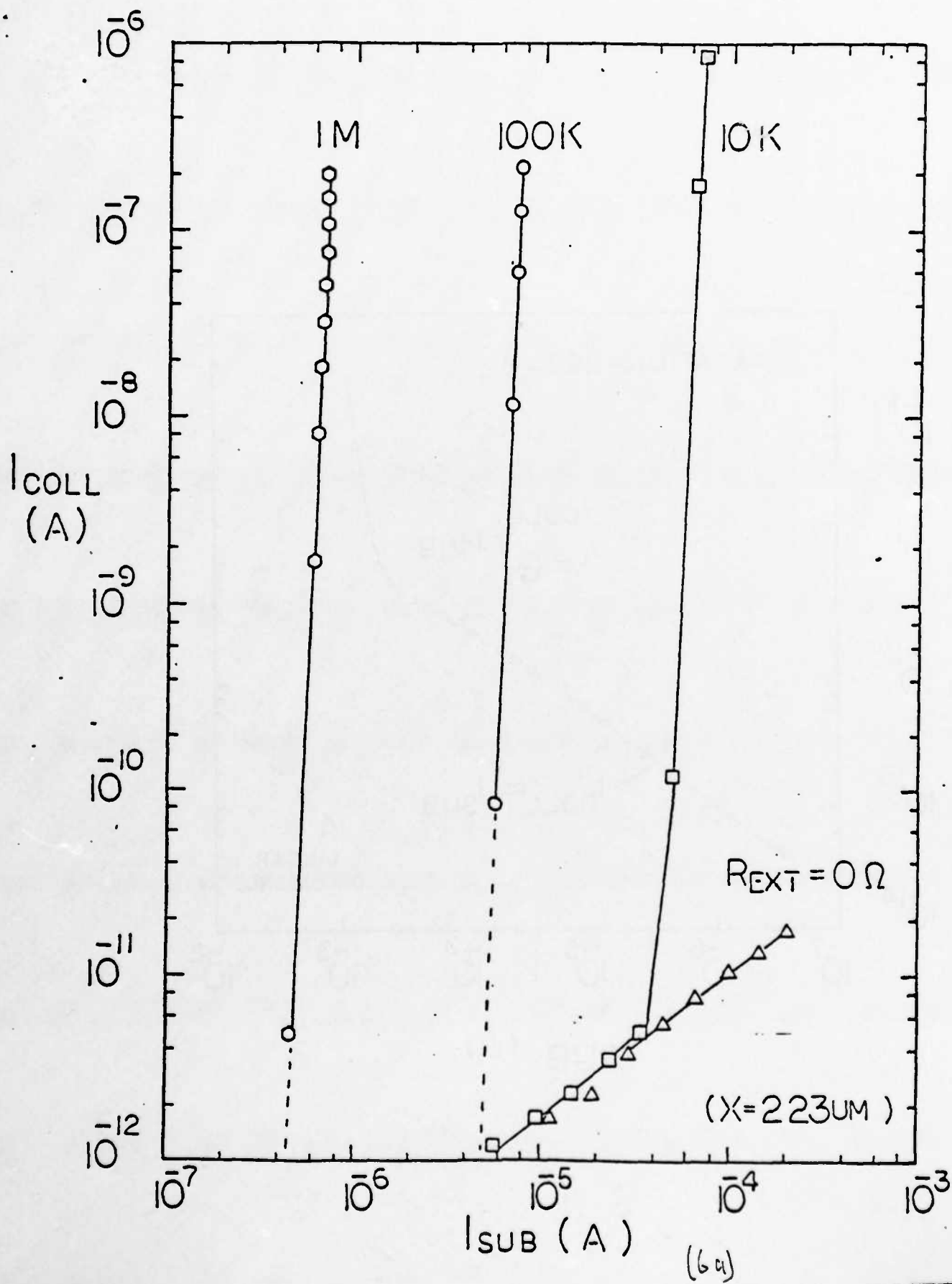
$I_{\text{COLL}} \propto I_{\text{SUB}}$

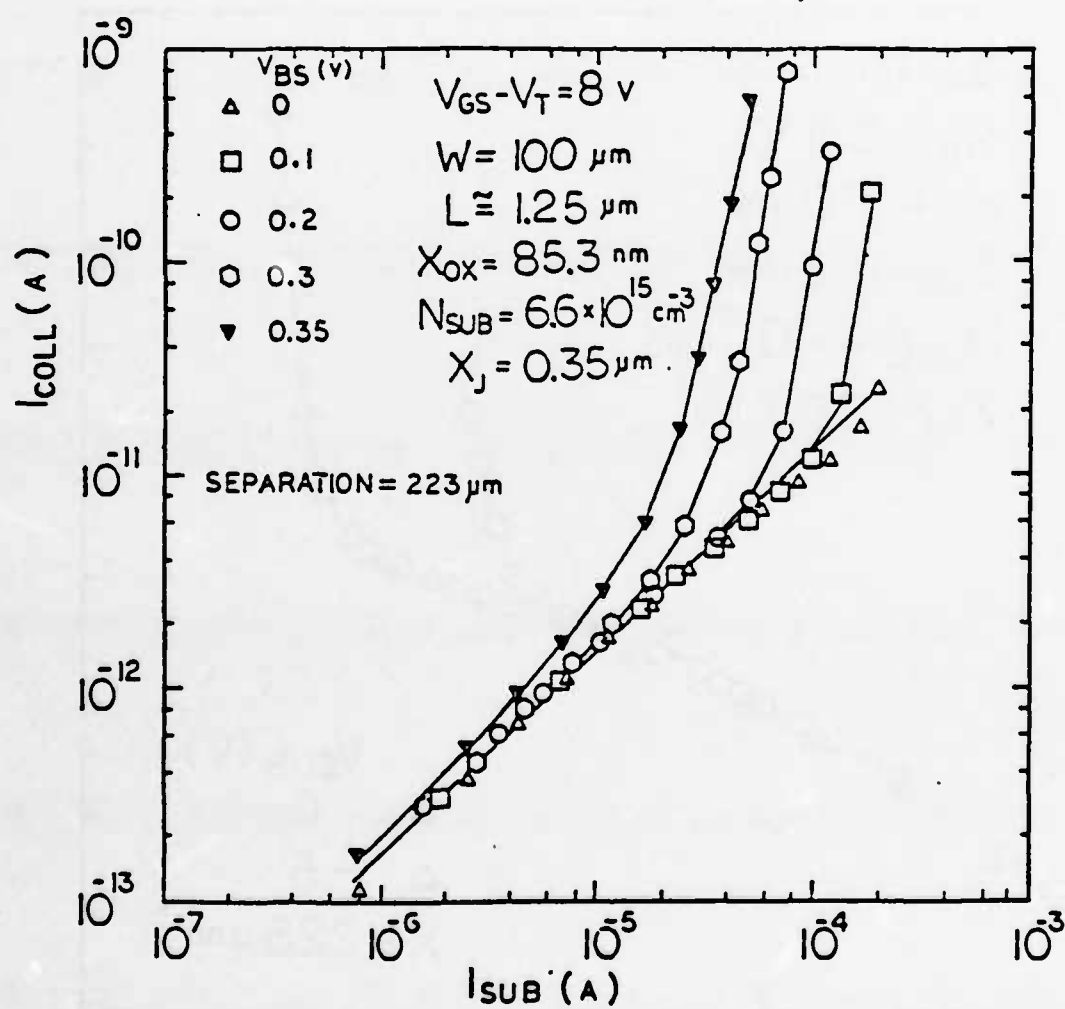
LINEAR  
DEPENDENCE

$I_{\text{SUB}} \text{ (A)}$

( 5 )

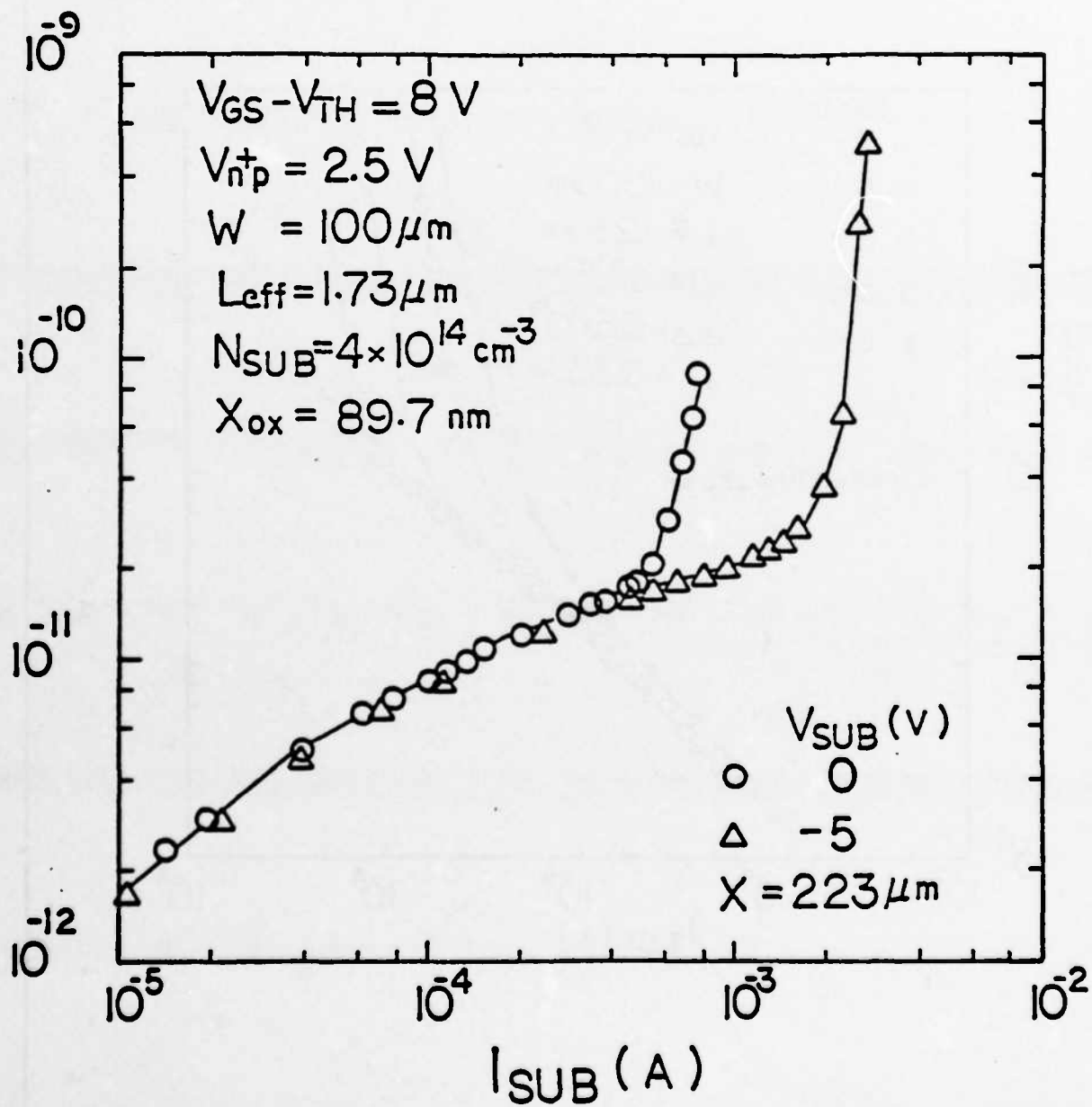






(b)

$I_{\text{COLL}}(\text{A})$



J2-3 (2.5)W LDR = 2.5  $\mu$ W

$V_{cc}$  (collector) = 5V

1/7/82, S.T.

configuration (A)

$I_e$  (A)

46 7400

LOGARITHMIC 3 X 3 CYCLES  
NEUTRAL A CENTER CO. 200000

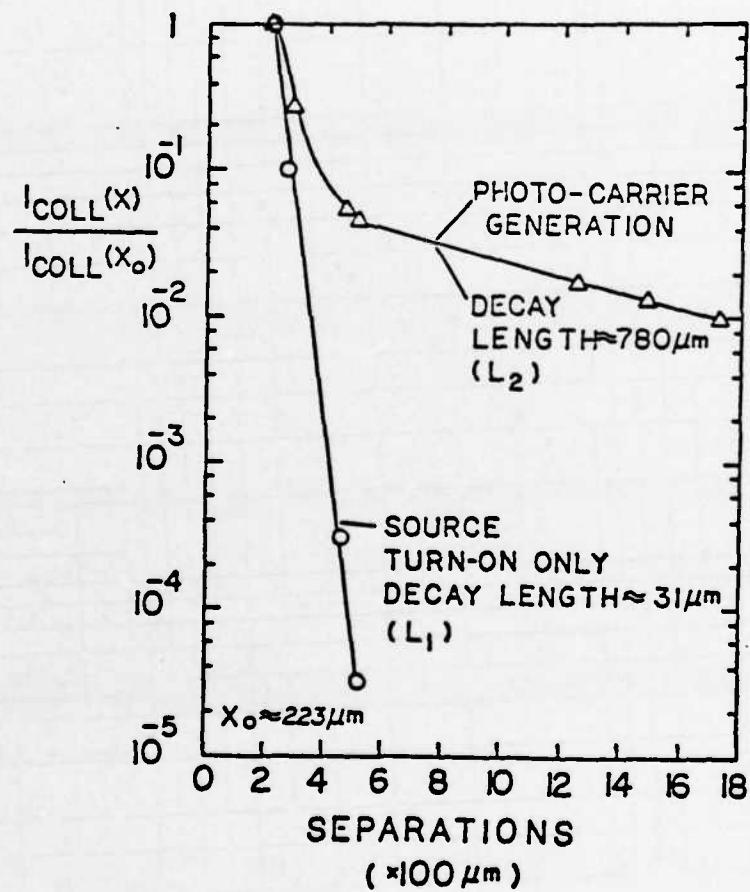
$V_{cc} - V_T$ (V)	$I_{JS}$ @ $V_{OS} = 0$ (V)
0	4
8	22.3 mA
12	33.3 uA

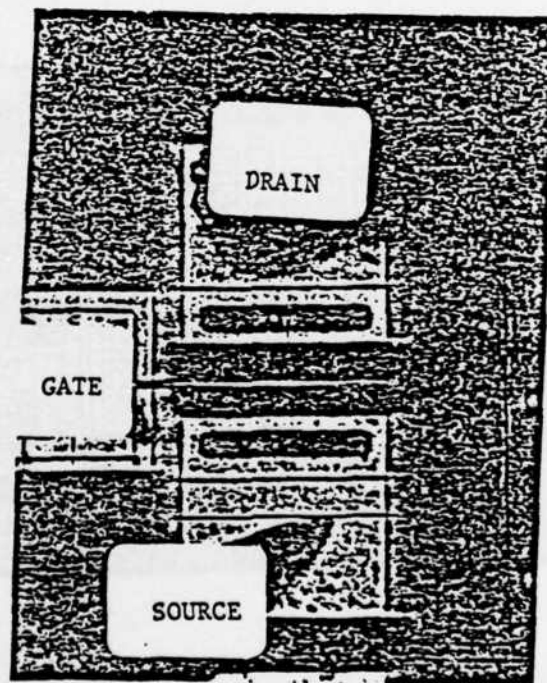
slope = 0.87

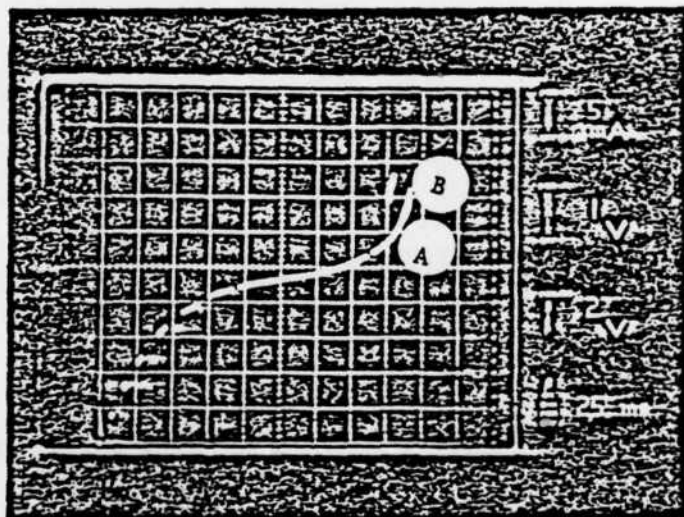
$I_{sub}$  (A)

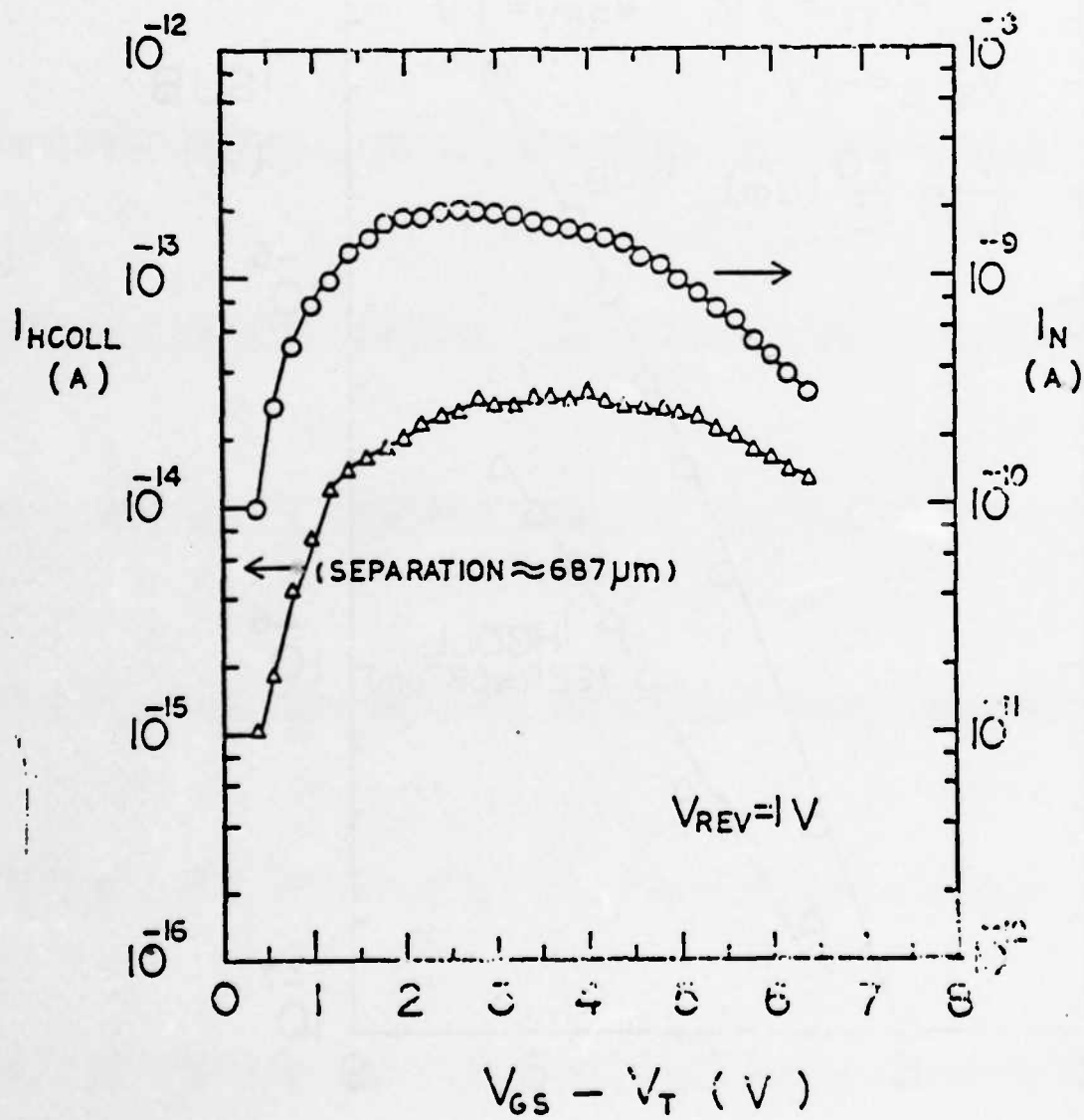
Fig 18)

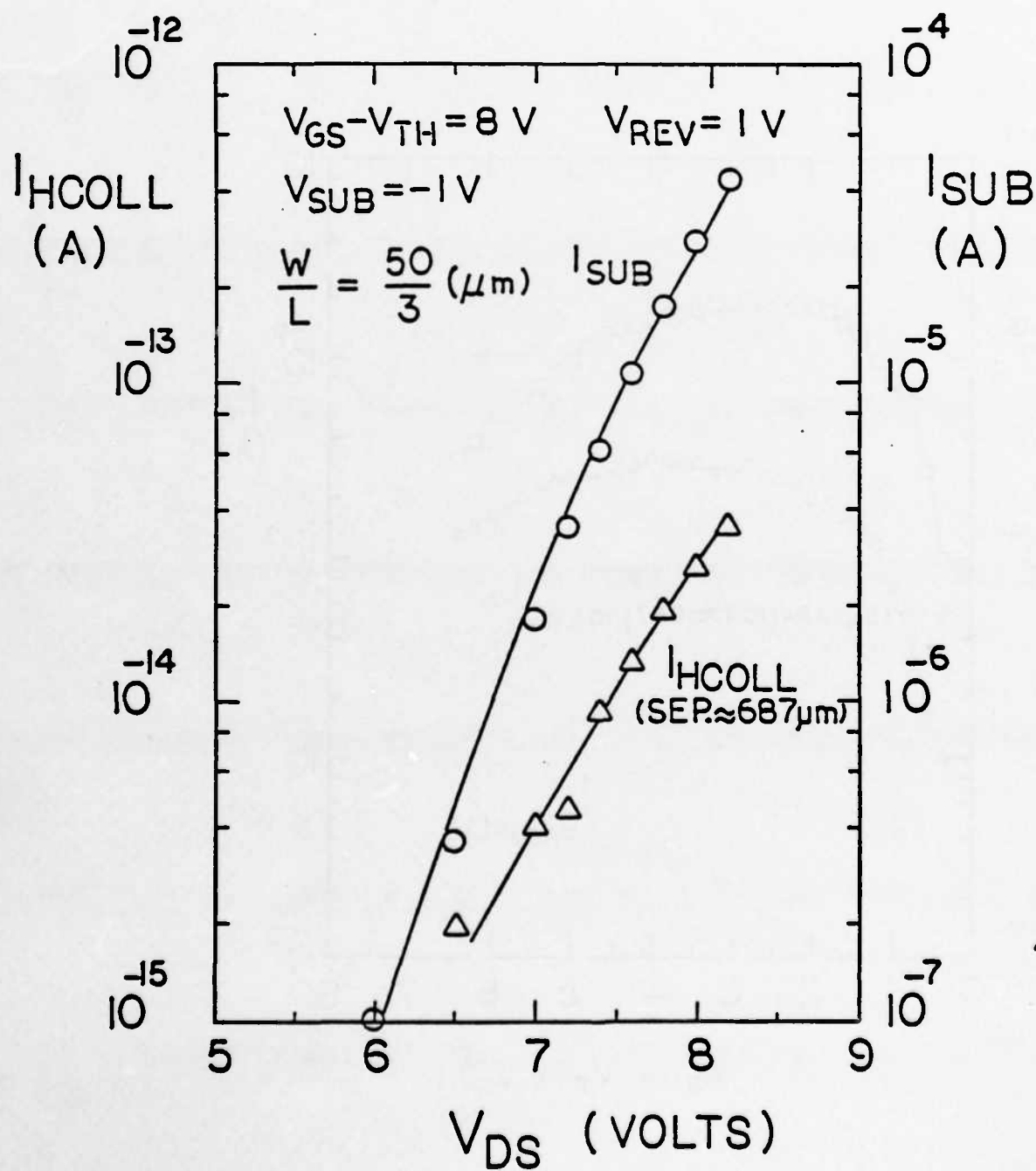






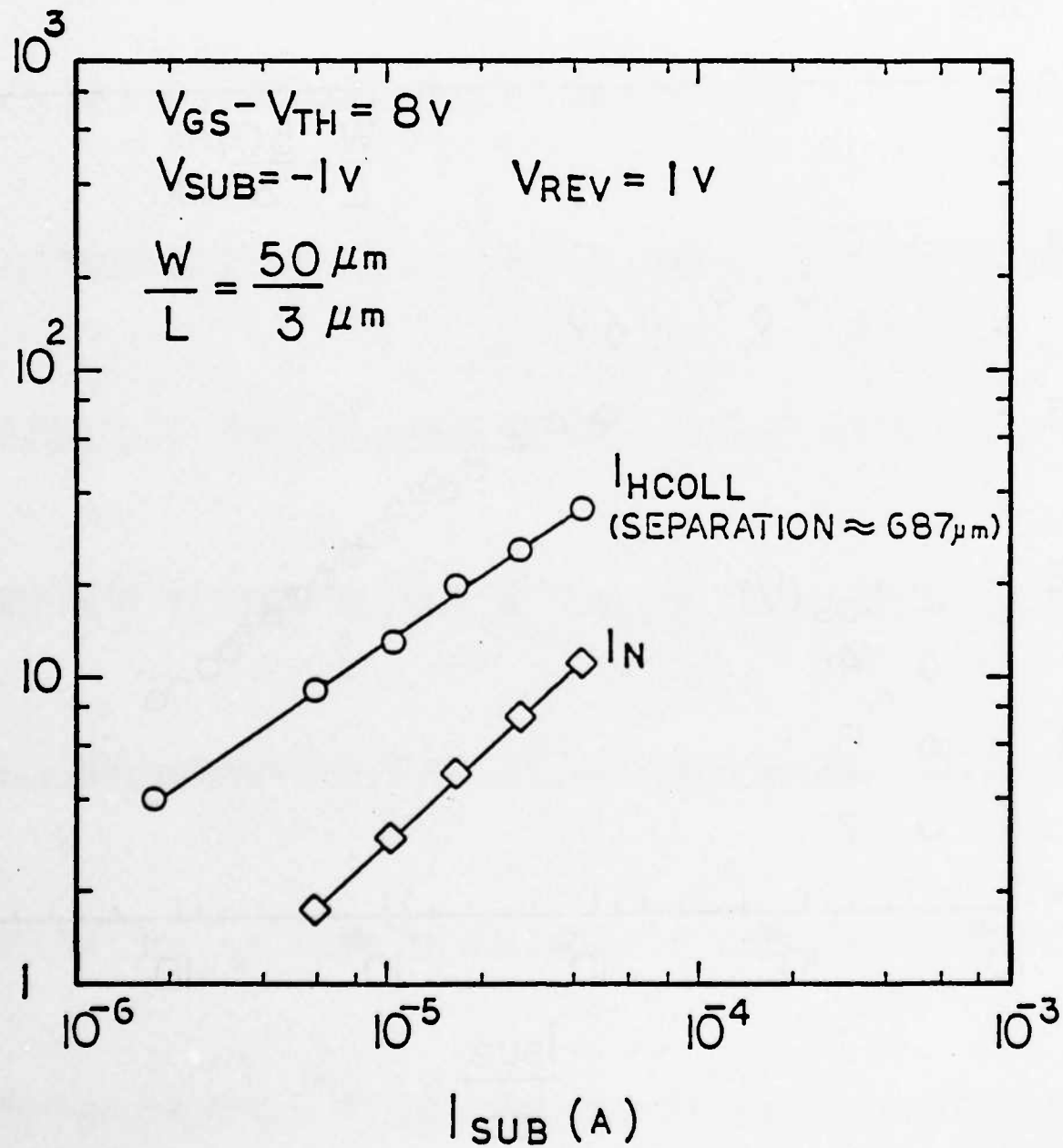




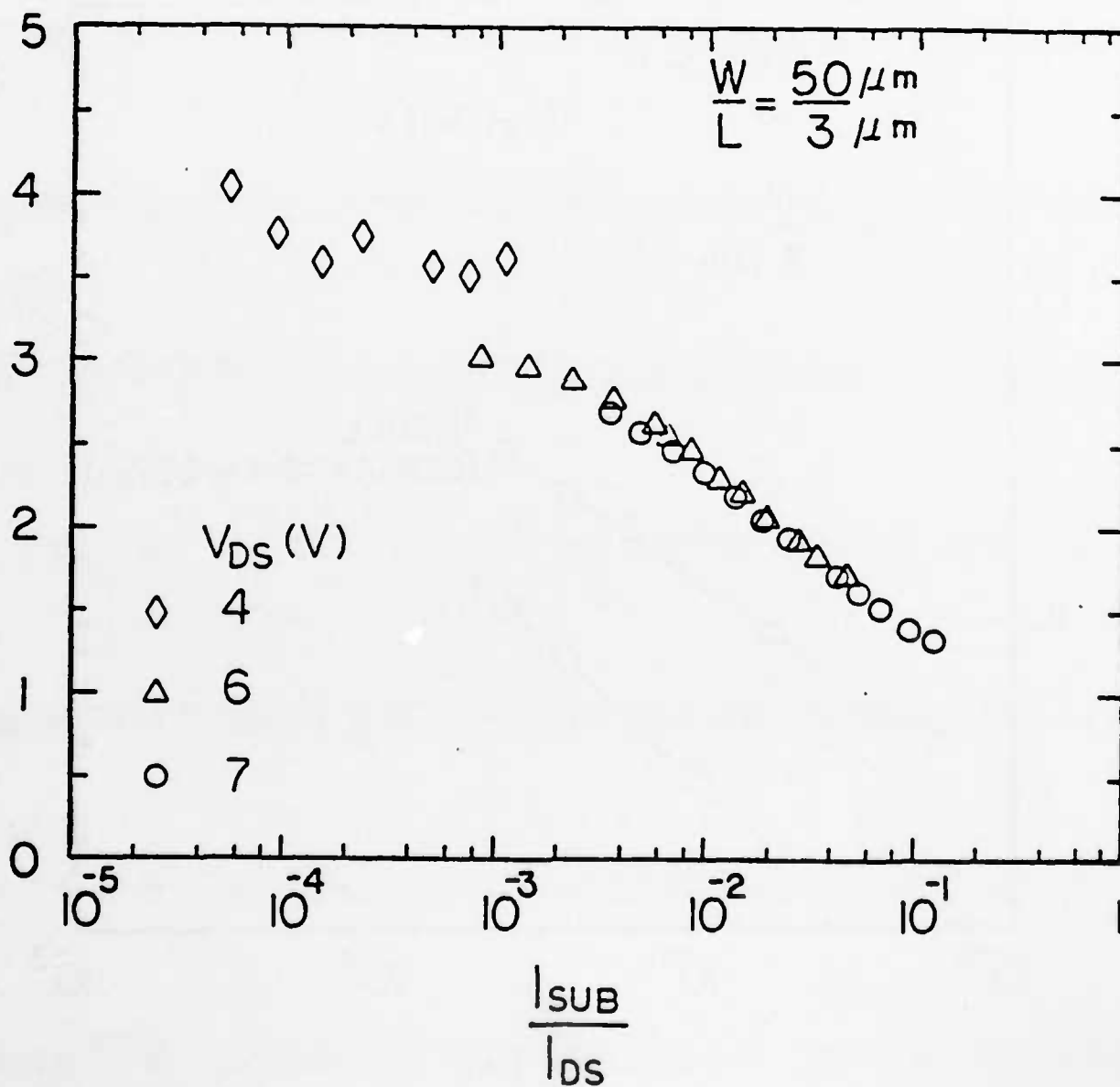


$I_N (10^{-10} \text{ A})$

$I_{HCOLL} (10^{-15} \text{ A})$



$$\frac{I_N}{I_{SUB}} (10^{-5})$$





# Photon Generation in Forward-Biased Silicon p-n Junctions

T.-C. ONG, K. W. TERRILL, S. TAM, AND C. HU, MEMBER, IEEE

**Abstract**—Photons are generated by forward biasing a silicon p-n junction at  $10^{-5}$ – $10^{-4}$  quantum efficiency through radiative recombination. At large distances from the forward-biased junction, leakage currents of magnitudes significant for some VLSI circuits can appear due to the substrate minority carriers generated by the photons. The effective decay length of the measured leakage current is about several hundred to one thousand micrometers. The effects of forward biasing an input node or a parasitic lateral bipolar transistor are, therefore, longer ranged than commonly assumed.

## I. INTRODUCTION

**L**IGHT EMISSION from silicon has long been observed in both forward-biased and reverse-biased avalanche silicon p-n junctions [1], but its effects in modern IC's has been little discussed. For forward-biased p-n junctions the mechanism for photon generation is radiative recombination, and the light emission has been used as a monitor of the uniformity of current in p-n-p-n thyristors [2]. In reverse-biased avalanche junctions, the mechanism for photo generation is direct transitions between different valence bands [3] or hot-electron/hole recombinations [1]. Fig. 1 shows the spectra for both a forward-biased and a reverse-biased avalanche junction [1]. The spectrum of the forward-biased case has a peak at 1.1 eV and a sharp cutoff at both high and low energies. For a reverse-biased avalanche junction, the spectrum is broad and extends to photon energies greater than 3 eV.

It was recently reported that photon generation induced by hot carriers does exist in silicon MOSFET's when operating in the saturation region [4]–[6]. The same phenomenon was also observed in CMOS VLSI devices [6]. The mechanism of photon generation and hence the spectrum are similar or identical to the case of a reverse-biased avalanche p-n junction [3], [6]. As device dimensions are scaled down, the effect of hot-electron-generated photons in silicon integrated circuits has become important since they can generate minority carriers in the substrate and discharge sensitive nodes. Already DRAM refresh time degradation due to this phenomenon is well known [7], and upset of SRAM or logic circuit is possible [8]. This paper describes the other occurrence of light emission, that from forward-biased p-n junctions [6], with the intention of highlighting its ability to discharge sensitive nodes in IC circuits.

Manuscript received August 15, 1983; revised October 11, 1983. This work was sponsored by DARPA under Grant N00039-81-K-0251.

The authors are with the Department of Electrical Engineering and Computer Sciences, Electronics Research Laboratory, University of California, Berkeley, CA.

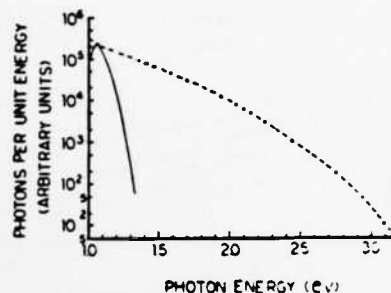


Fig. 1. The emission spectra for forward-biased and reverse-biased avalanche silicon p-n junctions. Solid line (—) for the forward-biased case, dashed line (---) for the avalanche breakdown case [1].

## II. EXPERIMENT

Our simple test chip consists of many isolated junctions. The junctions used are actually the source or drain of enhancement-mode MOSFET's. One junction acts as a collector of minority carriers and is reverse-biased by 1 V. The other junctions, which are separated from the collector by varying distances, act as injectors of minority carriers and are forward biased by a current source. Measurements are done for both  $n^+$  diffusions in a p-substrate and  $p^+$  diffusions in an n-substrate. The collecting junctions are  $49 \mu\text{m} \times 108 \mu\text{m}$  and  $0.3 \mu\text{m}$  deep in the p-substrate,  $25 \mu\text{m} \times 14 \mu\text{m}$  and  $0.6 \mu\text{m}$  deep in the n-substrate. The substrate concentrations used are  $6.6 \times 10^{15} \text{ cm}^{-3}$  for the p-substrate (boron doped) and  $2.7 \times 10^{17} \text{ cm}^{-3}$  for the n-substrate (phosphorus doped). There are no particular reasons to choose different substrate concentrations and no intention to make the concentrations differ so much. We chose these two samples only because they were available during the measurement. Fig. 2(a) and 2(b) show the collection current as a function of the distance between the injecting junction and the collecting junction for  $n^+p$  and  $p^+n$  diodes, respectively. The reverse saturation currents at 1-V reverse bias are 0.5 pA for the  $n^+p$  diode and 0.1 pA for the  $p^+n$  diode. Each has been subtracted from the measured collection current.

## III. RESULTS AND DISCUSSION

From Fig. 2, it can be seen that for small distances the collection current decreases rapidly, but for large distances the decrease in the collection current is much slower. The rapid decrease at small distances is explainable by diffusion of minority carriers. The approximate decay lengths for the minority carriers are 25 and  $12 \mu\text{m}$  for the p-substrate and the n-substrate, respectively. These are in good agreement with

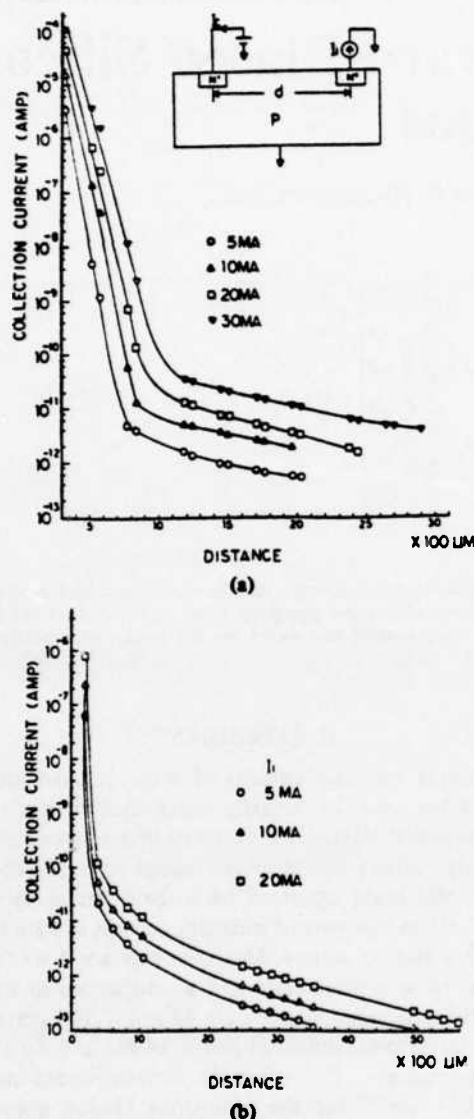


Fig. 2. (a) The collection current versus distance with the injecting current as a parameter for the  $6.6 \times 10^{15} \text{ cm}^{-3}$  p-type substrate. (b) For the  $2.7 \times 10^{17} \text{ cm}^{-3}$  n-type substrate.

reverse recovery-time measurement, which gives a lifetime of 230 ns for the p-type substrate and a lifetime of 100 ns for the n-type substrate. The slow decrease in the collection current at large distances is not explainable by diffusion theory, because carriers can not decay with one diffusion length for some distance and then with another much longer diffusion length. The slow decrease of the collection current is believed to be the result of photon generation of minority carriers. The photons themselves are generated through a radiative band-to-band recombination process as described in the Section I. They have energies near that of the band gap (see Fig. 1) thus having long absorption length. When the photons are absorbed, electron-hole pairs are created in the substrate and the electrons or holes can then be collected by a reverse-biased junction. In Fig. 2, beyond  $1000 \mu\text{m}$  (40 mil) the collection currents can be fitted with effective decay lengths of  $\sim 700 \mu\text{m}$  (p-substrate) and  $\sim 1100 \mu\text{m}$  (n-substrate).

The number of photons generated per second by a forward-biased junction is given by

$$N_p = \frac{I_i \eta}{q} \quad (1)$$

where  $\eta$  is the quantum efficiency, which is a function of the minority-carrier concentration,  $I_i$  is the injected current, and  $q$  is the charge of electron. Assuming there is no optical reflection from the top and bottom surfaces of the wafer, the number of electron-hole pairs generated per second per volume by these photons is

$$G(d) = \frac{N_p \alpha}{4\pi d^2} \exp(-\alpha d). \quad (2)$$

Here,  $d$  is the distance between the injector and the point where the electron-hole pairs are generated.  $\alpha$  is the absorption coefficient for these photons in silicon. The collection current is related to this generation rate by

$$I_c(d) = q V_c G(d) \quad (3a)$$

$$V_c \approx \frac{2\pi L_d (W/2 + L_d)(L/2 + L_d)}{3} \quad (3b)$$

$V_c$  is a "collection volume" dependent on collector size and carrier diffusion length but is assumed to be independent of  $d$ . The formula used for  $V_c$  is the volume of an ellipsoid with axes  $L_d$ ,  $W/2 + L_d$ , and  $L/2 + L_d$ .  $L_d$  is the diffusion length of minority-carriers and  $W$  and  $L$  are the width and length of the collecting junction, respectively. Equations (2) and (3a) are approximate formulas which are good when  $d$  is much larger than the injector and collector dimensions. To extract the  $\alpha$  in (2) we replot the data of p-substrate case in Fig. 3 as

$$R \equiv d^2 \frac{I_c}{I_i} = \frac{\eta V_c \alpha}{4\pi} \exp(-\alpha d) \quad (4)$$

versus the distance from the injecting junction. The figure is plotted for distances larger than  $700 \mu\text{m}$  where the effects of photons start to become important. At large distances  $R$  stays almost constant. This indicates that the collection current is originated from the absorption of photons which have a long absorption length. By extrapolating  $R$  back to  $d = 0 \mu\text{m}$ , the product of the absorption coefficient  $\alpha$  and the quantum efficiency  $\eta$  can be estimated with (4). Fig. 4 shows the measured  $\eta$  ( $\eta_e$ ), with  $\alpha$  used as an adjustable parameter to fit the theoretical quantum efficiency  $\eta_t$ , versus the injected current. Theoretically,  $\eta_t$  is equal to the ratio of the minority-carrier lifetime  $\tau$  to the radiative lifetime  $\tau_r$  [9]:

$$\eta_t = \frac{\tau}{\tau_r} = \tau B(n' + N_{sub}) \quad (5)$$

$n'$  is the excess minority-carrier concentration,  $N_{sub}$  is the substrate concentration, constant  $B$  is  $2 \times 10^{-15} \text{ cm}^3 \text{ s}^{-1}$  [10]. In (4), with  $V_c = 7.1 \times 10^5 \mu\text{m}^3$ ,  $\tau = 450 \text{ ns}$  for the

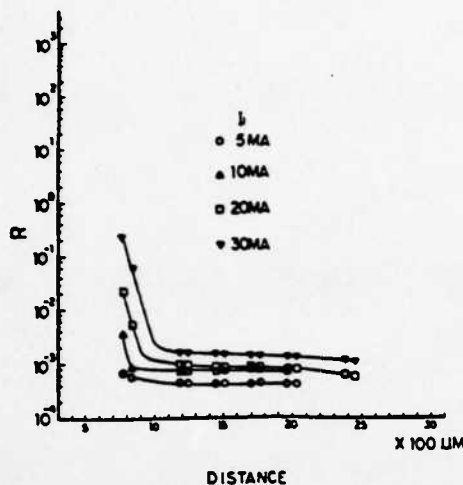


Fig. 3.  $R = d^2 I_c / I_i$  versus distance with the injection current as a parameter for the  $6.6 \times 10^{15} \text{ cm}^{-3}$  p-type substrate.

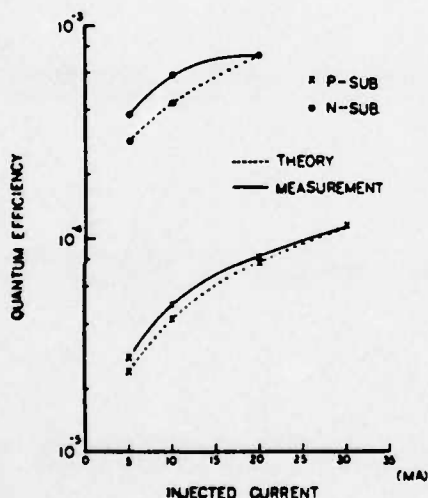


Fig. 4. Theoretical quantum efficiency and the quantum efficiency calculated from (4) versus the injected current (X) for the p-substrate, (O) for the n-substrate, solid line (—) for measurement, dashed line (---) for theory.

$n^+p$  junctions and  $V_c = 0.287 \times 10^5 \mu\text{m}^3$ ,  $\tau = 260 \text{ ns}$  for the  $p^+n$  junctions (from (3b)),  $\eta_e$  can have good agreement with  $\eta_i$  by setting  $\alpha = 2.5 \text{ cm}^{-1}$  for the p-substrate and  $\alpha = 4.5 \text{ cm}^{-1}$  for the n-substrate as shown in Fig. 4.  $\alpha = 2.5 \text{ cm}^{-1}$  is in reasonable agreement with the slopes at large distances in Fig. 3 and within the expected range based on the results by Vavilov [11].  $\alpha = 2.5 \text{ cm}^{-1}$  is also close to the reported data by Spitzer and Fan [12]. This result supports our argument about photon generation in forward-biased silicon p-n junctions.

#### IV. CONCLUSION

A forward-biased silicon p-n junction not only injects minority carriers into the silicon substrate but also generates photons through radiative recombination. At small distances from the injecting junction, the measured collection current is dominated by the diffusion transport of carriers. At large distances from the injector, the collection current is mainly

due to the substrate minority carriers that are generated by the absorption of photons. The collection current appears to have an effective decay length of about several hundred to one thousand micrometers when the effect of photons dominates. During the radiative recombination process, it takes approximately  $10^4 \sim 10^5$  electron-hole recombinations to generate one photon. The absorption coefficients are  $2.5 \text{ cm}^{-1}$  (p-substrate) and  $4.5 \text{ cm}^{-1}$  (n-substrate) in the two samples studied. The current generated by these photons can have a magnitude much larger than the reverse saturation current of the collecting junction.

The implications of photon generation in the silicon substrate may be illustrated with dynamic RAM circuits. In the undesirable case that any of the junctions in the peripheral circuit is forward biased by a substrate-current-induced voltage drop [13] or by positive voltage glitches on the input and output lines, errors can be induced deep inside the array because of the long decay length of the collection current. The guard ring and the epi-substrate techniques may not be as effective as expected in protecting nodes far from the I/O circuits.

#### ACKNOWLEDGMENT

Appreciation goes to the memory design manager of National Semiconductor Corporation, T. Klein, who supplied information about the DRAM bit map.

#### REFERENCES

- [1] A. G. Chynoweth and K. G. McKay, "Photon emission from avalanche breakdown in silicon," *Phys. Rev.*, vol. 102, no. 2, p. 369, 1956.
- [2] P. Voss, "Infrared observation of the breakdown behavior of high voltage p-n junctions and p-n-p structures in silicon," *IEEE Trans. Electron Devices*, vol. ED-20, p. 299, 1973.
- [3] W. Haecker, "Infrared radiation from breakdown plasmas in Si, GaSb, and Ge: Evidence for direct free hole radiation," *Phys. Stat. Sol.(a)*, vol. 25, p. 301, 1974.
- [4] S. Tam, F. C. Hsu, P. K. Ko, C. Hu, and R. S. Muller, "Hot-electron induced carriers in MOSFET's," *IEEE Electron Device Lett.*, vol. EDL-3, p. 376, Dec. 1982.
- [5] P. A. Childs, W. Eccleston, and R. A. Stuart, "Alternative mechanism for substrate minority carrier injection in MOS devices operating in low level avalanche," *Electron. Lett.*, vol. 17, pp. 281-282, 1981.
- [6] G. Gerosa, S. Stern, B. Bastani, and R. Chwang, "Quantification of photon generation in CMOS VLSI structure," presented at 41st annual Device Research Conference, Vermont.
- [7] B. Eitan, D. Frohman-Bentchkowsky, and J. Shappir, "Holding time degradation in dynamic MOS RAM by injection-induced electron currents," *IEEE Trans. Electron Devices*, vol. ED-28, p. 1515, 1981.
- [8] J. Matsunaga, H. Momose, H. Iizuka, and S. Kohyama, "Characterization of two step impact ionization and its influence in NMOS and PMOS's VLSI's," presented at IEDM (Washington, DC), Dec. 1980.
- [9] S. M. Sze, *Physics of Semiconductor Devices*, 2nd ed. New York: Wiley, 1981.
- [10] J. I. Pankove, *Optical Processes in Semiconductors*. Englewood Cliffs, NJ: Prentice-Hall, 1971.
- [11] V. S. Vavilov, "The absorption of free charge carriers by infrared radiation in silicon," *Sov. Phys. Solid State*, vol. 3, no. 2, pp. 346-349, Aug. 1960.
- [12] W. Spitzer and H. Y. Fan, "Infrared absorption in n-type silicon," *Phys. Rev.*, vol. 108, pp. 268-271, 1957.
- [13] F.-C. Hsu, R. S. Muller, and C. Hu, "A simplified model of short-channel MOSFET characteristics in the breakdown mode," *IEEE Trans. Electron Devices*, vol. ED-30, p. 571, 1983.

# LUCKY-ELECTRON MODEL OF CHANNEL HOT ELECTRON INJECTION IN MOSFETs

*S. Tam, P.K. Ko\*, and C. Hu*

Department of Electrical Engineering and Computer Sciences

and the Electronic Research Laboratory

University of California, Berkeley, CA 94720

\* Bell Laboratories, Holmdel, N.J. 07733.

## ABSTRACT

The lucky electron concept is successfully applied to the modelling of channel hot electron injection in n-channel MOSFETs, although the result can be interpreted in terms of electron temperature as well. This results in a relatively simple expression that can quantitatively predict channel hot electron injection current in MOSFETs. The model is compared with measurements on a series of n-channel MOSFETs and good agreement is achieved. In the process, new values for many physical parameters such as hot-electron mean-free-path are determined. Of perhaps even greater practical significance is the quantitative correlation between the gate current and the substrate current that this model suggests. The dominant hot-electron scattering mechanism is due to optical-phonons.

# LUCKY-ELECTRON MODEL OF CHANNEL HOT ELECTRON INJECTION IN MOSFETs

*S. Tam, P.K. Ko<sup>\*</sup>, and C. Hu*

Department of Electrical Engineering and Computer Sciences  
and the Electronic Research Laboratory  
University of California, Berkeley, CA 94720

<sup>\*</sup> Bell Laboratories, Holmdel, N.J. 07733.

## 1. Introduction

Recent extensive studies on short-channel MOSFETs have made much progress in the development as well as the understanding of the limitations of VLSI circuits [1,2]. One important aspect of the physics of the short-channel MOSFETs is the injection of hot-electrons from the channel into the gate [3,4,5]. Channel hot-electron injection (CHEI) into the gate can result in the degradation of device performance due to the trapping of electrons in the gate oxide and the generation of interface traps. The phenomenon of channel hot-electron injection is also widely used as the programming mechanism in EPROMs. Therefore a simple quantitative model of the CHEI effect in MOSFETs would be useful for the understanding and design of future VLSI devices. At high enough drain voltages, CHEI can be measured directly as gate currents [3]. Indirect measurements down to very small currents utilizing floating gate MOSFETs have also been made [5] although results from these measurements are more difficult to interpret due to the complex device structures [6].

There are two separable parts in the task of modelling CHEI. The first part



is to find the electric field, particularly the maximum field, in the channel. A more reliable means of finding the field are 2-D or 3-D device computer simulations [7,8,9] although good accuracy has also been reported for an analytical field model based on pseudo-two-dimensional considerations [10]. The second part is to model CHEI in terms of the channel electric field. The second part is the subject of the present paper. This is most often done using the concept of electron temperature. However no reliable theory or experiments has yet been developed relating the field and the electron temperature [11]. Consequently, all CHEI models have been either empirical in nature [3] or computationally complex and untested [12].

In this paper, we expand on and make a more thorough presentation of a physical model for CHEI in MOSFETs based on the lucky-electron concept [13]. Direct measurements of channel hot-electron injection as MOSFET gate current is used to evaluate the model. Studies are made on polysilicon-gate MOSFETs with arsenic doped source/drain regions. The model to be presented assumes that the maximum channel field in the direction of the channel current is known. Experimentally, in this paper, the maximum field is deduced from the measured substrate current. A by-product of this study is the reaffirmation, by theory and experiments, of a correlation between the gate current and the substrate current (at least when the gate voltage is higher than the drain voltage) [14].

## 2. Model

The streaming or lucky electron approach of modelling the hot-electron distribution was originated by Shockley [15]. Later Verwey et. al. [16] used this approach in their study on substrate hot electron injection in MOSFETs which was later refined and verified by Ning et. al. [17]. Hu [13] modified the substrate lucky electron injection model and applied it to CHEI in MOSFETs.

Conceptually, the lucky electron model of CHEI can be described as follows. In order for channel hot-electrons to be reach to the gate, the hot-electrons must gain sufficient kinetic energy from the channel field and has its momentum redirected elastically to surmount the potential barrier at the silicon/silicon-dioxide interface. The momentum of these hot-electrons must then be re-directed elastically toward the silicon/silicon-dioxide interface. To quantify the probability that these electrons could eventually be collected by the gate, several types of inelastic scatterings have to be considered ( figure(1) ). From point A to B, a channel electron gains energy from the channel field and becomes "hot". At B, re-direction of the hot-electron takes place. From point B to C, ( C is situated at the interface ), the hot-electron must not suffer any energy-robbing collision so that it will retain the energy required to surmount the silicon/silicon-dioxide potential barrier. The hot-electron must also suffer no collision in the oxide image-potential well located between C and D. Once the hot-electron arrives at location D, it will be swept toward the gate electrode by the aiding field. In the following, we shall analyze mathematically these various processes involved separately.

## 2.1. Probability of Acquiring Sufficient Normal Momentum

A schematic illustration of the injected hot electron in the potential-distance space is shown in figure(2). In order for the hot-electron to surmount the silicon/silicon-dioxide potential barrier (  $\phi_b$  in volts ), its kinetic energy must be greater than  $\phi_b$ . To acquire the kinetic energy  $\phi_b$ , the hot-electron will have to travel a distance  $d$  where  $d = \frac{\phi_b}{E_x}$  if we assume the electric field  $E_x$  to be constant. The probability of a channel electron to travel a distance  $d$  or more without suffering any collision can be written as  $[3] e^{-\frac{d}{\lambda}}$ , where  $\lambda$  is the scattering mean-free-path of the hot electron. The interpretation of  $\lambda$  will be discussed



later. ( $\lambda$  depends on the optical-phonon scattering mean-free-path and the impact-ionization mean-free-path.) Hence, we can write  $e^{-\frac{\phi_b}{E_x \lambda}}$  as the probability that an electron will acquire kinetic energy greater than the silicon/silicon-dioxide potential barrier.

If an electron is to be emitted, its momentum must be re-directed toward the silicon/silicon-dioxide interface by an elastic scattering and has a sufficiently large momentum component perpendicular to the interface. An electron that possesses exactly the energy  $\phi_b$  will be emitted only if its momentum is directed into an infinitesimally small solid angle normal to the Si/SiO<sub>2</sub> interface. Assuming isotropic re-direction scatterings, an electron possessing energy ( $\phi = \phi_b + \Delta\phi$ ) would, due to geometrical consideration only has the probability to surmounting the barrier [18] as,

$$(1/2) \left[ 1 - \sqrt{\frac{\phi_b}{\phi_b + \Delta\phi}} \right] \approx \frac{\Delta\phi}{4\phi_b} \quad (1)$$

where  $\frac{\Delta\phi}{4\phi_b}$  is correct for  $\Delta\phi \ll \phi_b$ . The probability of an electron to have the kinetic energy between  $\phi_b + \Delta\phi$  and  $\phi_b + \Delta\phi + d(\Delta\phi)$  is

$$\frac{d e^{-\frac{(\phi_b + \Delta\phi)}{E_x \lambda}}}{d(\Delta\phi)} d(\Delta\phi) = \frac{e^{-\frac{(\phi_b + \Delta\phi)}{E_x \lambda}}}{E_x \lambda} d(\Delta\phi) \quad (2)$$

The probability of an electron having enough normal momentum to surmount the silicon/silicon-dioxide potential barrier can be evaluated by integrating the product of equations(1) and (2) over all  $\Delta\phi$ . Then the probability of an electron acquiring the required kinetic energy and retaining the appropriate momentum after redirection can be expressed as,

$$P_{\phi_b} = \int_{\Delta\phi=0}^{\Delta\phi=\infty} \frac{\Delta\phi}{4\phi_b} e^{-\frac{(\phi_b + \Delta\phi)}{E_x \lambda}} \frac{d(\Delta\phi)}{E_x \lambda} = 0.25 \frac{E_x \lambda}{\phi_b} e^{-\frac{\phi_b}{E_x \lambda}} \quad (3)$$

## 2.2. Probability of Collision-Free Travel to the Barrier Peak

We now evaluate the probability that a hot electron travels to the Si-SiO<sub>2</sub> interface without suffering any collision,  $P_1$ , after undergoing a re-directing collision at varying depths below the interface. Here,  $P_1$  is a scattering probability factor weighted by the electron concentration in the inversion layer. If we have  $n(y)$  as the electron concentration at depth  $y$  and position  $x$  in the channel ( see figure(3) ), then  $P_1$  can be expressed as

$$P_1 = \frac{\int_{y=0}^{\infty} n(y) e^{-\frac{y}{\lambda}} dy}{\int_{y=0}^{\infty} n(y) dy} \quad (4)$$

The exponential term in equation(4) is the probability of not suffering any energy-robbing collisions as described previously. In order to find  $n(y)$ , we need to find the potential  $\varphi(y)$  by solving the Poisson equation.

$$\frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} = -\frac{q}{\epsilon_{Si}} (N_{sub} + n) \quad (5)$$

By assumming strong inversion and the gradual channel approximation (

ie.  $\frac{\partial E_y}{\partial y} \gg \frac{\partial E_x}{\partial x}$ ) equation(5) can be solved ( see Appendix(1) ) and we have

$$P_1 = 1 - \alpha e^{\alpha} E_1(\alpha) \quad (6)$$

where

$$\alpha = \frac{6kT}{q\lambda E_{ox}} = \frac{5.172 \times 10^{-4} T}{E_{ox} \lambda}$$

$$E_{ox} \approx \frac{(V_{gs} - V_{ds})}{X_{ox}}$$

$T$  is the absolute temperature, and  $E_1$  is the exponential integral. In short channel devices, we need to include the  $\frac{\partial E_x}{\partial x}$  term in equation(5) when  $V_{ds}$  is sufficiently high. Appendix(2) shows how this can be done approximately.

The last probability factor we have to consider is the scattering in the oxide

image-potential well. Here, the probability  $P_2$  can be written as [18]

$$P_2 = e^{-\frac{y_0}{\lambda_{0z}}} \quad (7)$$

where

$$y_0 = \sqrt{\frac{q}{16 \cdot \pi E_{0z} \epsilon_{0z}}}$$

and  $\lambda_{0z} = 3.2nm$  [19]. Combining the constants, we arrive at the expression for  $P_2$  as

$$P_2 = e^{\frac{-300}{\sqrt{E_{0z}}}} \quad (8)$$

where  $E_{0z}$  is in V/cm.

We shall define the product of  $P_1$  and  $P_2$  as  $P$  which is essentially only a function of  $E_{0z}$ .  $P(E_{0z})$  can be approximated by ( ref.[14] and Appendix(2) )

$$P(E_{0z}) \approx \left[ \frac{5.66 \times 10^{-6} E_{0z}}{(1 + \frac{E_{0z}}{1.45 \times 10^5})} \times \frac{1}{(1 + \frac{2 \times 10^{-3}}{L_{eff}} e^{\frac{-E_{0z} X_{0z}}{1.5}})} + 2.5 \times 10^{-2} \right] e^{\frac{-300}{\sqrt{E_{0z}}}} \quad (9)$$

for  $E_{0z} \geq 0$  and

$$P(E_{0z}) \approx 2.5 \times 10^{-2} e^{\frac{-X_{0z}}{\lambda_{0z}}} \quad (9a)$$

for  $E_{0z} < 0$ .

### 2.3. Evaluation of the CHEI Gate Current

We can now express the gate current  $I_{gate}$  in terms of the probabilities described in the previous sections, as

$$I_{gate} = I_{ds} \int_0^L P_{\phi_s} P(E_{0z}) \frac{dx}{\lambda_r} \quad (10)$$

where  $L$  is the length of the channel ( ie. from the source to the drain metallurgical junction or even beyond ) ,  $\lambda_r$  is the re-direction scattering mean free path.

The factor  $\frac{dx}{\lambda_r}$  can be interpreted as the probability of re-direction over  $dx$ .

Hence, the integral in equation(10) gives the total probability of CHEI. The

parameter  $\lambda_r$  will be discussed later.

Since the probability  $P_{\phi_b}$  depends exponentially on  $E_z$  and  $E_z$  also varies exponentially with  $x$  [10,20], the integrand in equation(10) is a sharply peaking function. Therefore, an approximate expression for the integral is

$$I_{gate} \approx I_{ds} \frac{\Delta L}{\lambda_r} [P_{\phi_b} P(E_{ox})]_{\max} \quad (11)$$

where  $\Delta L$  is the length of the region of significant CHEI. In the case of  $V_g > V_d$ ,

$[P_{\phi_b} P(E_{ox})]_{\max}$  occurs at the drain where  $P_{\phi_b}$  or  $E_z$  is maximum.  $\Delta L$  may be

approximated by  $\frac{P_{\phi_b}}{\frac{dP_{\phi_b}}{dx}}$  evaluated at  $x=L$ . ( Note that  $dP(E_{ox})/dx$  is usually

much smaller than  $dP_{\phi_b}/dx$  . )

Equation (10) is approximately,

$$I_{gate} = 0.25 \frac{\lambda^2 E_m^3 I_{ds} P(E_{ox}|L)}{\lambda_r \phi_b^2 (dE_z/dx)|_L} e^{-\frac{\phi_b}{E_m \lambda}} \quad (12)$$

$$\approx 0.5 \times \frac{I_{ds} X_{ox}}{\lambda_r} \left( \frac{\lambda E_m}{\phi_b} \right)^2 P(E_{ox}|L) e^{-\frac{\phi_b}{E_m \lambda}} \quad (12a)$$

where  $E_m$  is the channel electric field at the drain end and we have used

$[dE_z/dx]_L \approx \frac{E_m}{2X_{ox}}$  in equation(12a) [20].

#### 2.4. Correlation to Substrate Current

Substrate current in the MOSFET is due to the impact ionization of the hot electrons in the drain high field region. Since the hot electrons responsible for CHEI and those responsible for the substrate current are heated by the same field, there should be a correlation between these two processes. This correlation has been studied and verified. The substrate current is known to be related to  $E_m$  by [14].

$$I_{sub} = I_{ds} \frac{A_i E_m^2}{B_i (dE_z/dx)|_L} e^{-\frac{B_i}{E_m}} \quad (13)$$

where  $A_i$  and  $B_i$  are the pre-exponential and exponential constants in the ionization coefficient [21]. Eliminating  $E_m$  from the exponential term in equation (12) using equation (13),

$$\frac{I_{gate}}{I_s} = C \cdot P(E_{ox}) \cdot \left( \frac{I_{sub}}{I_s} \right)^{\frac{\Phi_b}{B_i \lambda}} \quad (14)$$

where

$$C = 0.25 \frac{\lambda^2}{\lambda_r \Phi_b^2} \left( \frac{dE_z}{dx} \right)_L \left( \frac{\Phi_b}{B_i \lambda} \right)^{-1} \left( \frac{B_i}{A_i} \right)^{\frac{\Phi_b}{B_i \lambda}} E_m^{3 - \frac{2\Phi_b}{B_i \lambda}} \quad (15)$$

$$\approx 0.25 \frac{\lambda^2}{\lambda_r \Phi_b^2} \left( \frac{B_i}{A_i} \right)^{\frac{\Phi_b}{B_i \lambda}} E_m^{2 - \frac{\Phi_b}{B_i \lambda}} (2X_{ox})^{1 - \frac{\Phi_b}{B_i \lambda}} \quad (15a)$$

$$\approx K \cdot X_{ox}^{1 - \frac{\Phi_b}{B_i \lambda}}$$

The approximate  $\frac{dE_z}{dx}$  used in obtaining equation(12a) is used in obtaining equation(15a). The constant  $K$  has a value of approximately  $1.5 \times 10^{-9}$  using the results presented in section(4) and assuming that  $E_m = 2 \times 10^5 \text{ Vcm}^{-1}$ . The slope in the log-log plot of  $\frac{I_{gate}}{I_s}$  versus  $\frac{I_{sub}}{I_s}$  will be equal to  $\frac{\Phi_b}{B_i \lambda}$ . Since  $B_i$  and the hot-electron scattering mean-free-path  $\lambda$  are independent of  $E_{ox}$ , the slope provides a method to determine the barrier height  $\Phi_b$  and its dependence on  $E_{ox}$ . This subject will be discussed in section(4).

### 3. Experimental Results and Discussion

Experiments on CHEI are carried out on a series of n-channel polysilicon gate MOSFETs where the processing parameters of the test devices are listed in Table(1). The test transistors have a channel width of  $100 \mu\text{m}$ . In the experiments, the gate current ( $I_{gate}$ ), substrate current ( $I_{sub}$ ), and source current ( $I_s$ ) were measured simultaneously. The resolution of the gate-current measurement was approximately 1 fA.

Figure(4) is a typical plot of CHEI gate current as a function of the gate-to-source voltage ( $V_{gs}$ ) at constant drain-to-source voltage ( $V_{ds}$ ). The impact-ionization substrate current is also shown. The familiar bell shaped curves are observed. Qualitatively, the dependence of the gate current on  $V_{gs}$  and  $V_{ds}$  can be explained as follows. The channel electric field in the MOSFET is proportional to the difference between  $V_{ds}$  and the drain saturation voltage ( $V_{dsat}$ ) [10]. At low  $V_{gs}$ ,  $V_{dsat}$  is small and hence the channel electric field at the drain is high. However, due to the fact that at low  $V_{gs}$ , the oxide electric field at the drain end is in a direction that will inhibit the collection of the hot-electrons by the gate electrode, we expect no measurable gate current coming from the portion of the channel with  $V_{ch} > V_g$ . When  $V_{gs}$  increases towards  $V_{ds}$ , the oxide field near the drain end becomes more favorable and we see a sharp increase in gate current. When  $V_{gs}$  increases further,  $V_{dsat}$  will also increase (although at a slower rate due to the velocity saturation effect [10]) and the peak channel electric field decreases, so the bell shaped curve of the gate current results (channel-field limited regime).

In figure(5), the measured gate current for the test devices are shown against  $V_{ds}$ . The gate-to-source voltage is fixed at 10 volts. The dependence of the gate current on the channel length is apparent. Reduction of the channel length reduces  $V_{dsat}$ . Therefore for the same drain-to-source voltage, the channel electric field, and hence  $I_{gate}$ , is higher in shorter channel devices. The devices with thinner oxide thickness (figure(5b)) has higher gate current because the channel-electric field is higher.

Also observed is the flattening-off and in some cases, a decrease of the gate current when  $V_{ds}$  approaches and eventually exceeds  $V_{gs}$ . This is the same electrode limited behavior we described in the last paragraph. Figure(6a) and (6c) illustrates the band diagrams for the channel-field limited and the electrode-

limited regimes. When  $V_{ds}$  is smaller than  $V_{gs}$ , the oxide field at the point of maximum channel electric field (ie. the drain end) is in a direction favorable to the collection of the injected electrons by the gate electrode. When  $V_{ds}$  is equal to  $V_{gs}$ , the oxide field is zero at the drain end of the channel. At this or higher  $V_{ds}$ , the hot-electrons at the drain becomes increasing more difficult to reach the gate electrode and the gate current will be dominated by injection at locations closer to the source where the channel field is weaker but the oxide field is still favorable. Therefore, when  $V_{ds}$  is greater than  $V_{gs}$ , we would expect the observed gate current to flatten. Since there will be a finite probability that some hot-electrons will be trapped in the gate oxide at the peak injection point, the oxide field at the peak injection point will decrease rapidly and the peak injection point will move further toward the source. These chain of events may eventually lead to a decrease in gate current when  $V_{ds}$  exceeds  $V_{gs}$ . This indeed is observed in our experimental results. In figure(7), we have shown the dependence of measured  $I_{gate}$  versus time. Although the biasing voltages are fixed, we observe that  $I_{gate}$  decreases with increasing time. This decrease can be attributed to the trapping of hot-electrons at the peak injection point just described.

#### 4. Analysis of Experimental Results

In the previous section, we have presented the measured gate current for a number of devices. In order to compare our experimental results with the lucky electron " model , we shall focus on the normalized gate current where this quantity is defined as the ratio of the gate current to the source current ( ie.  $\frac{I_{gate}}{I_s}$  ) of the MOSFET. The source current is essentially equal to the drain current, differing only by  $I_{sub}$ . This is demonstrated in figure(8) where we have replotted the data presented in figure(5a). The solid and the dash lines in figure(8) are the calculated  $\frac{I_{gate}}{I_s}$  which will be discussed later. In view of equa-



tion(10), the normalized gate current can be interpreted as the total probability of CHEI. In figure(9), we have plotted the constant normalized gate current contour (ie. the  $L_{eff}$  and  $V_{ds}$  at constant  $\frac{I_{gate}}{I_s}$ ).

#### 4.1. Potential Barrier $\Phi_b$ and Correlation to $I_{sub}$

The first parameter we shall consider is the effective potential barrier between the silicon conduction band edge and the silicon-dioxide conduction band edge. This potential barrier has been found to be [17]

$$\Phi_b = 3.2 - \beta \sqrt{E_{ox}} - \vartheta E_{ox}^{\frac{2}{3}} \quad V \quad (16)$$

The quantity 3.2 V is the Si-SiO<sub>2</sub> interface barrier. The second term in equation(16) represents the barrier lowering effect due to the image field [18]. The last term in equation(16) accounts phenomenologically for the finite probability of tunneling between the silicon and the silicon dioxide [17]. For SiO<sub>2</sub>,  $\beta = 2.59 \times 10^{-4} (V \cdot cm)^{\frac{1}{2}}$ . The parameter  $\vartheta$  will be determined by comparison with our experimental results.

Figure(10) is the log-log plot of the normalized gate current versus the normalized substrate current using the gate-to-drain voltage  $V_{gd}$  as the parameter. For a constant  $V_{gd}$ , the oxide electric field  $E_{ox}$  at the location of maximum  $E_x$  (ie. the drain), is constant (parasitic drain diffusion resistance may affect  $E_{ox}$  slightly). As discussed in section(2), the slope in this plot gives the quantity  $\frac{\Phi_b}{B_i \lambda}$ . In figure(11), the experimentally determined slopes are plotted against the calculated  $E_{ox}$  (effect of drain diffusion resistance included) using the data in figure(10). A best fit to the dependence of  $\Phi_b$  on  $E_{ox}$  is obtained by selecting  $\vartheta$  so that the calculated  $B_i$  and  $\lambda$  product will have the minimum variance over all  $E_{ox}$  considered. By adopting this procedure, we choose  $\vartheta$  to be  $4 \times 10^{-3} V^{\frac{1}{3}} cm^{\frac{2}{3}}$ . This is in contrast with the value  $1 \times 10^{-3} V^{\frac{1}{3}} cm^{\frac{2}{3}}$  suggested by

Ning et. al. [17]. The product  $E_i\lambda$  then has the value of 1.24V and a variance of ~2.7% over all  $E_{ox}$ . This value of  $E_i\lambda$  is then fixed in all subsequent calculations. The theoretical curves shown in figure(10) are based on equation(14) where a value of  $A_4 = 1.6 \times 10^5 \text{ Vcm}^{-1}$  is assumed. The dependence of  $\bar{v}_y$  on  $E_{ox}$  is also illustrated in figure(11).

#### 4.2. Comparison of $I_{gate}$ Measurements and Model

With the silicon/silicon-dioxide potential barrier determined, we are ready to make direct comparisons between the lucky electron model and experimental results. In figure(8), we have shown the theoretical gate current curves calculated by integrating equation(11) numerically ( solid lines ). The channel-electric field is determined from a simple quasi-2D MOSFET model formulated by Ko [10]. The effective re-direction mean-free-path  $\lambda_r$  and the hot-electron scattering mean-free-path,  $\lambda$  are the only fitting parameters in the calculation of  $\frac{I_{gate}}{I_s}$ . The fit is insensitive to the re-direction mean-free-path  $\lambda_r$ , which is chosen to be  $61.6 \text{ nm}$  based on theoretical consideration as discussed in section(5). With this, we now have  $\lambda$ , the hot-electron scattering mean-free-path as the only fitting parameter. A value of  $9.2 \text{ nm}$  for  $\lambda$  gives the best fit for all channel lengths considered. We have also included the calculated gate current curves using the approximate analytical expression presented in equation(14) ( dash lines ). Good agreements are obtained between the theoretical model and the experimental results.

In figures(12a) and (12b), we have used the correlation between the gate current and the substrate current that was presented in section(2.4) (equation(14)) to calculate the normalized gate current from measured  $I_{sub}$ . The solid lines are calculations based on equations(14&15) while the dash lines are calculations based on equations(14&15a). In all the calculations, the  $E_i$  and  $\lambda$  product

of 1.24V is used. The values of  $\lambda$  and  $\lambda_r$  used here are the same as those used in the calculation shown in figure(10). The agreement between experimental results and calculations is very good.

In figure(13), we have shown the calculated normalized gate current curve for the data shown in figure(4). A direct numerical integration approach is used here. The bell shaped dependence of the normalized gate current on  $V_{gs}$  is obtained. There is some discrepancies between the calculated and the experimental results at the high gate current regime. This probably is due to the effects of electron trapping on the oxide field at the peak injection point that was discussed in section(3).

#### 4.3. Effects of Temperature

In figure(14), we have shown the effect of temperature on the CHEI gate current. The CHEI gate current decreases with increasing lattice temperature. The temperature coefficient ( T.C. =  $\frac{d(\ln(I_{gate}/I_s))}{dT}$  ) is experimentally found to be  $-0.025\text{ }^\circ\text{C}^{-1}$  ( ie. doubling of  $I_{gate}$  for every  $\sim 29^\circ\text{C}$  decrease in temperature). This agrees with a previous report [4] where a T.C. of  $-0.0299\text{ }^\circ\text{C}^{-1}$  was obtained. The temperature dependence of the substrate current is also shown in figure(14). The temperature coefficient for the substrate current is  $-0.0132\text{ }^\circ\text{C}^{-1}$  which is smaller than that for the gate current. (Ref.[4] obtained a substrate current T.C. of  $-0.01036\text{ }^\circ\text{C}^{-1}$ .) The decrease in the CHEI gate current is due to the reduction in hot-electron scattering mean-free-path ( $\lambda$ ). The decrease in the impact ionization substrate current with temperature is due to the reduction in optical-phonon mean free path (  $\lambda_{op}$  ) [25]. From equation(14), one expects the two slopes to differ by a factor of  $\frac{\Phi_b}{B_1\lambda} \approx 2.1$ , in good agreement with the temperature coefficients given above. The eventual increase of  $I_{gate}$  and  $I_{sub}$  at high temperatures is due to the increase of the intrinsic thermal

generation of carriers which results in substrate hot electron injection. In view of the analytical expression for CHEI gate current presented in equation(12), we see that the predominant temperature dependence of CHEI comes in through  $\lambda$  in the exponential form. As a first order approximation and for the same biasing condition, we expect the maximum channel electric field  $E_m$  and  $I_s$  to be independent of temperature. The ratio of the normalized gate current at two temperatures is

$$\frac{\left[ \frac{I_{gate}}{I_s} \right]_{T_1}}{\left[ \frac{I_{gate}}{I_s} \right]_{T_0}} \approx \left( \frac{\lambda_{T_1}}{\lambda_{T_0}} \right)^2 e^{\frac{-\phi_b}{E_m} \left( \frac{1}{\lambda_{T_1}} - \frac{1}{\lambda_{T_0}} \right)} \quad (17)$$

In obtaining equation(17), we have assumed the effects of the temperature dependence of  $\lambda_{ox}$  and  $\lambda_r$  to be small when compared to the effects due to  $\lambda$ . This is a relatively good assumption because  $\lambda$  enters equation(12) in an exponential. If the scattering of the hot-electrons are dominated by scattering due to optical-phonons, then we shall expect the temperature dependence of  $\lambda$  to be similar to that of optical phonon scattering mean-free-path [25],

$$\lambda_{op}(T) = \lambda_0 \tanh\left(\frac{E_p}{2kT}\right) \quad (18)$$

where  $\lambda_0$  is the hot-electron scattering mean-free-path as  $T$  approaches  $0^\circ K$  and  $E_p$  is the optical-phonon energy. In figure(15), we have shown the change in the normalized gate current versus temperature obtained from the data presented in figure(14). The temperature dependence of our experimental results corresponds very well with that of the optical-phonons. A value of 9.2 nm is used for the  $\lambda$  at room temperature ( ie.  $33^\circ C$  ) and good agreement is obtained when  $E_p = 0.070$  eV. From equation(17),  $\lambda$ , is calculated to be 10.6 nm. This is in excellent agreement with the 10.5 nm determined by Ning et al. [17] but larger than the 7.6 nm reported by Sze [25]. For comparison, we have also included the experimental results obtained from Matsumoto et al. [4] in figure(15). The

excellent agreement between the simple approximation shown in equation(17) and the experimental results indicated that the principal energy losing mechanism is due to optical phonons scatterings. Again the deviation between data and theory at high temperature is believed to be due to substrate hot electron injection [17].

## 5. Discussion

We shall in this section, try to interpret the physical meanings of  $\lambda$ , the channel hot-electron scattering mean-free-path and  $\lambda_r$ , the momentum re-direction mean free path. A way to estimate the channel impact-ionization threshold energy ( $E_I$  in eV) will be presented.

As pointed out by many authors [12,14,16,22], the channel hot-electron scattering mean-free-path ( $\lambda$ ) depends on the optical-phonon scattering (m.f.p.  $=\lambda_p$ ) and the impact-ionization (m.f.p.  $=\lambda_I$ ) mean-free-paths. We see that when the hot-electron kinetic energy (K.E.) is less than  $E_I$ , the predominant scattering mechanism will be due to optical-phonons. When the K.E. of the hot-electrons become larger than  $E_I$  (ie.  $E_I \leq \text{K.E.} \leq \Phi_b$ ), both the optical-phonon scattering and the impact-ionization scattering will be significant. In this energy range, the effective scattering m.f.p. ( $\lambda^*$ ) should be formulated as  $\frac{1}{\lambda^*} = \frac{1}{\lambda_p} + \frac{1}{\lambda_I}$ . Due to Verwey et al. [16], we have the probability for a channel hot-electron to acquire the energy  $\Phi_b$  to be

$$\text{Prob.}(\Phi_b) \sim e^{\frac{-E_I}{qE_s\lambda_p}} e^{\frac{-(\Phi_b - \frac{E_I}{q})}{E_s\lambda^*}} \quad (19)$$

By letting  $\frac{E_I}{q} = \gamma\Phi_b$ , this probability can be written as

$$\text{Prob.}(\Phi_b) \sim e^{\frac{-\Phi_b}{E_s}(\frac{1}{\lambda_p} + \frac{(1-\gamma)}{\lambda_I})} \quad (20)$$

By comparing the above equation with the formulation presented in section(2.1),

we can readily obtain

$$\frac{1}{\lambda} = \frac{1}{\lambda_p} + \frac{1-\gamma}{\lambda_I} \quad (21)$$

Therefore, equation(21) implies that the channel hot-electron scattering m.f.p. is a combination of  $\lambda_p$  and  $\lambda_I$  where the influence of impact-ionization is weighted by  $(1-\gamma)$ . This is different to the common belief that  $\lambda^{-1} = \lambda_p^{-1} + \lambda_I^{-1}$  [12,22]. In view of this and the large value of  $\lambda_I (\approx 70nm)$ , it may be concluded that the dominant hot-electron inelastic scattering at the channel of the MOSFET is due to optical-phonons. This further justify the assumption we made in section(4.3) when formulating the temperature dependence of  $\lambda$ . Using the result that  $B_i \lambda = 1.24V$  and  $\lambda = 9.2nm$ , we have  $B_i = 1.34 \times 10^5 Vcm^{-1}$ . This is believed to be more accurate than many values found in the literature that ranges between  $1.08 \times 10^5 Vcm^{-1}$  and  $1.75 \times 10^5 Vcm^{-1}$  [23-26].

The momentum re-direction mean-free-path has been interpreted as the m.f.p. of scatterings where momentum relaxation takes place without significant energy exchange. As pointed out by Long [27], the two most likely candidates for elastic scatterings in Si are intervally acoutical-phonon scattering ( g-scatterings ) and long-wavelength acoutical-phonon scattering. In our calculations, the value of  $\lambda_r$  is set to the combined m.f.p. for the long wavelength acoutic phonon and the intervally acoutic phonon scattering which is equal to 61.6 nm as obtained by Duh et al. [23].

In our discussion on the correlation between the C.H.E.I. gate current and the impact-ionization substrate current, the slope in the  $\log(\frac{I_{gate}}{I_s})$  versus  $\log(\frac{I_{sub}}{I_s})$  plot yields the quantity  $\frac{\Phi_b}{B_i \lambda}$ . If we interpret the  $e^{\frac{-B_i}{E_x}}$  in the impact-ionization coefficient as  $e^{\frac{-E_I}{qE_x \lambda_p}}$ , then the quantity  $\frac{\Phi_b}{B_i \lambda}$  may be re-written as  $\frac{q \Phi_b}{E_I} \frac{\lambda_p}{\lambda} = \frac{1}{\gamma} \frac{\lambda_p}{\lambda} \approx \frac{1}{\gamma}$ . In the last step, we have used equation(21) and assumed

$\lambda_I \gg \lambda_p$ . ( $\lambda_I$  was determined by Schockley [15] to be  $88nm$ , by Troutman [22] to be  $40nm$  and by Verwey [16] to be  $187nm$ .) Experimentally, the slope,  $\frac{\phi_b}{E_i \lambda}$ , and hence  $\gamma$ , may be determined and we obtained  $\gamma=0.429$ . This in turn implies that  $E_f$  equals to  $1.23eV$ .

## 6. Reconciliation between the Lucky Electron Model and the Effective Temperature Model

The CHEI model presented in this study is based on the lucky electron concept. There is, however, another approach which is based on the effective hot-electron temperature [3,11,29]. In the effective temperature  $T_e$  approach, the CHEI gate current can be formulated as

$$Prob.(\phi_b) \sim e^{-\frac{q\phi_b}{kT_e}} \quad (22)$$

where  $k$  is the Boltzman constant and  $T_e$  is the effective hot-electron temperature. A recent experimental study suggested that on very short-channel MOS-FETs, the pure ballistic argument used in the lucky electron concept probably is less accurate than the quasi-thermal equilibrium effective temperature approach [11]. Nevertheless, the lucky electron based hot-electron model was found to be able to describe the hot-electron phenomena well. From this, the empirical relationship of  $T_e = \frac{q}{k} E_x \lambda = 1.07 \times 10^{-2} E_x$  is proposed [11].

At this point, we like to point out that the basic lucky electron energy consideration is used mainly to derive the probability  $P_{\phi_b}$  in section(2.1). The formulation on the other probabilities (ie.  $P_1$  and  $P_2$ ) are quite independent on the lucky electron energy consideration and therefore are still valid. The only modification to the model, if the effective electron temperature concept is used, is  $P_{\phi_b}$ , which will be

$$P_{\phi_b} = 0.25 \times \frac{kT_e}{q\phi_b} e^{-\frac{q\phi_b}{kT_e}} \quad (23)$$



## 7. Conclusion

We have presented a quantitative model for the channel hot-electron injection in MOSFETs. The model is based on the lucky electron concept, although the result can be interpreted in terms of electron temperature as well. Three probabilities are derived to describe the physical mechanisms responsible for CHEI gate current. They are, (i) probability of a hot-electron to gain enough kinetic energy and normal momentum, (ii) probability of not suffering any inelastic collision during transport to the Si-SiO<sub>2</sub> interface, and (iii) to suffer no collision in the oxide image-potential well. An important result of this study is the reaffirmation, by theory and experiments, of a correlation between the gate current and the substrate current. The theoretical model agrees well with experimental results.

Based on experimental results and theoretical consideration, the dependence of Si-SiO<sub>2</sub> effective barrier height on the oxide electric field is evaluated. A coefficient that accounts phenomenologically for the finite probability of tunneling is determined to be  $4 \times 10^{-5} V^{\frac{1}{3}} cm^{\frac{2}{3}}$ . This is in contrast with the value of  $1 \times 10^{-5} V^{\frac{1}{3}} cm^{\frac{2}{3}}$  obtained by Ning et al. [17].

From the temperature dependence of CHEI, we concluded that optical-phonon scattering is the dominant inelastic scatterings suffered by the channel hot-electrons. The effect of impact-ionization mean-free-path on the hot-electrons is found to be less significant. A value of  $9.2 nm$  for the hot-electron scattering mean-free-path at  $33^{\circ}C$  is obtained. Based on the temperature dependence study, we found that an optical-phonon-energy of  $70 meV$  best describe our results and also agrees with the data presented by Matsumoto et al. [4]. From this, we determined that  $\lambda_p$  equals to  $10.6 nm$  which is in excellent agreement with that obtained by Ning et al. [17].

The momentum re-direction scattering in the channel that is necessary for CHEI can be characterized by the momentum re-direction mean-free-path  $\lambda_r$ . Based on theoretical consideration, we attribute this to be the combined m.f.p. for intervalley acoustical-phonon scattering and the long-wavelength acoustical-phonon scattering ( *m.f.p.* = 51.6nm ). However, the model presented in this paper is insensitive to the exact value of  $\lambda_r$ .

From the detailed consideration on the impact-ionization coefficient and the hot-electron scattering mean-free-path, we derived an impact-ionization energy of 1.23eV and  $E_i$  to be  $1.34 \times 10^8 \text{ Vcm}^{-1}$ .

### 8. Acknowledgement

The support of the HP Design-Aids Groups, Cupertino, CA is gratefully acknowledged. Research sponsored in part by the DARPA grant N00039-81-K-0251 and AFOSR grant F49620-79-C-0718.

### Appendix(1) : Derivation of $P_i$ in strong inversion

At strong inversion, the potential along the y-direction ( vertical ) can be expressed as ,

$$-\frac{d\varphi}{dy} = E_y \approx \frac{\sqrt{2}kTn_i}{qL_D N_{sub}} e^{\frac{\beta_T \varphi}{2}} \quad (A1.1)$$

where  $\varphi$  is the potential,  $n_i$  is the intrinsic carrier concentration,  $L_D$  is the intrinsic Derby length and  $\beta_T = \frac{q}{kT}$ . By integrating equation (A1.1) from the surface (  $y=0$  ) to  $y$ , we obtain

$$e^{\beta_T \varphi} = \frac{1}{\left[ e^{-\beta_T \varphi/2} + \frac{n_i y}{\sqrt{2}L_D N_{sub}} \right]^2} \quad (A1.2)$$

By inserting the result from equation (A1.2) into equation (4), we obtain equation (6).

## Appendix (2) : Derivation of $P_1$ for Short Channel Devices

In short channel devices, we have to take into account the two dimensional effects in order to achieve reasonable results. This can be accomplished approximately by separating the electron concentration ( $n$  in equation(4) ) into two components, namely  $n_{ox}$  and  $n_d$ . The component  $n_{ox}$  is the mobile charge controlled by the gate while the component  $n_d$  is the mobile charge controlled by the drain. Therefore equation(4) becomes,

$$P_1 = \frac{\int_{y=0}^{\bar{y}} n_{ox}(y) e^{-y/\lambda} dy + \int_{y=0}^{\bar{y}} n_d(y) e^{-y/\lambda} dy}{\int_{y=0}^{\bar{y}} n(y) dy} \quad (A2.1)$$

The denominator in equation(A2.1) can be expressed as,

$$N_{mobile} = \int_{y=0}^{\bar{y}} n(y) dy = \frac{I_{ds}}{q W v_{sat}} \quad (A2.2)$$

where  $v_{sat}$  is the saturation velocity of the channel electrons and  $W$  is the device width. The charge controlled by the gate can be approximated as

$$N_{ox} = \int_{y=0}^{\bar{y}} n_{ox} dy = \frac{C_{ox}'}{q} (V_{gs} - V_{ds}) \quad (A2.3)$$

where  $C_{ox}'$  is the gate capacitance per unit area and  $(V_{gs} - V_{ds})$  is the voltage across the oxide at the drain end. Using the result from the strong inversion case ( equation(6) ), the first integral in the numerator of equation(A2.1) is

$$\int_{y=0}^{\bar{y}} n_{ox}(y) e^{-y/\lambda} dy = \left[ 1 - \alpha e^{\alpha} E_1(\alpha) \right] \frac{C_{ox}'}{q} (V_{gs} - V_{ds}) \quad (A2.4)$$

If we approximate the charge component  $n_d$  to be constant from  $y=0$  to a critical depth  $Y_m$ , then the second integral in the numerator of equation(A2.1) becomes,

$$\int_{y=0}^{\bar{y}} n_d(y) e^{-y/\lambda} dy \approx \int_{y=0}^{Y_m} n_d(y) e^{-y/\lambda} dy = n_d \lambda \left( 1 - e^{-\frac{Y_m}{\lambda}} \right) \quad (A2.5)$$

and

$$Y_m = \frac{N_{mobile} - N_{ox}}{n_d} \quad (A2.6)$$

where  $n_d = \frac{\epsilon_{Si}}{q} |(dE_x/dx)|_{E_m}$ . Combining the results from above, we have

$$P_1 = \frac{\left[1 - \alpha e^{\alpha} E_1(\alpha)\right] N_{ox} + n_d \lambda \left(1 - e^{-\frac{Y_m}{\lambda}}\right)}{N_{mobile}} \quad (A2.7)$$

In the case when  $V_{gs} \leq V_{ds}$ , we shall assume that at the drain end, the drain controls all the mobile charge the substrate charge (mobile charge no longer  $\gg$  than substrate charge) and the charge at the gate that are associated with the drain (ie.  $(n_d - N_{sub})Y_m = N_{mobile} + N_{ox}$  where  $N_{ox} = \frac{C_{ox}}{q}(V_{ds} - V_{gs})$ ). Then the probability  $P_1$  may be approximated as

$$P_1 \approx \frac{\lambda}{Y_m} \left(1 - e^{-\frac{Y_m}{\lambda}}\right) \approx \frac{\lambda}{Y_m} \quad (A2.8)$$

The oxide field at the drain end for this case is in a direction that opposes the motion of the injected electrons. The probability factor  $P_2$  then becomes

$$P_2 = e^{\frac{-X_{ox}}{\lambda_{ox}}} \quad (A2.9)$$

The silicon/silicon-dioxide potential barrier is increased to  $\phi_b = 3.2 + (V_{ds} - V_{gs})$  (in volts).

The expression for  $P_1$  in equation(A2.7) requires the knowledge of the channel-field gradient. A reasonable fit to  $P_1$  when  $E_{ox} \geq 0$  [14] is

$$P_1 \approx \frac{5.88 \times 10^{-6}}{\left(1 + \frac{E_{ox}}{1.45 \times 10^8}\right)} \times \frac{1}{\left(1 + \frac{2 \times 10^{-3}}{L_{eff}} e^{-\frac{E_{ox} X_{ox}}{1.5}}\right)} + 2.5 \times 10^{-2} \quad (A2.10)$$

and equals to  $2.5 \times 10^{-2}$  when  $E_{ox} < 0$ .

## References

- (1) H. N. Yu, A. Reisman, C. M. Osburn, and D. L. Critchlow, "1  $\mu m$  MOSFET VLSI technology : Part 1 - an overview," IEEE Solid-State Circuits, vol. SC-14, pp.240-246, 1979.

- (2) T.H. Ning, P.W. Cook, R.H. Dennard, C.M. Osburn, S.E. Schuster and H.N. Yu, "1  $\mu\text{m}$  MOSFET VLSI technology: Part 5 - Hot-electron design constraints," IEEE Trans. Electron Devices, vol. ED-26, p. 346, 1979.
- (3) P.E. Cortell, R.R. Troutman, and T.H. Ning, "Hot-electron emission in n-channel IGFETs," IEEE Trans. Electron Devices, vol. ED-23, pp.520-533, 1979.
- (4) H. Matsumoto, K. Sawada, A. Asai, M. Hirayama, and K. Nagasawa, "Effects of long-term stress on IGFET degradation due to hot electron trapping," IEEE Trans. Electron Devices, vol. ED-28, p.923, 1981.
- (5) B. Eitan and D. Frohman-Bentchkowsky, "Hot-electron injection into oxide in n-channel MOS devices," IEEE Trans. Electron Devices, vol. ED-23, p.328, 1981.
- (6) F.H. Gaensslen and J.M. Aiken, "Sensitive techniques for measuring small MOS gate currents," IEEE Electron-Device Letters, vol. EDL-1, pp.231-233, 1980.
- (7) T. Toyabe, K. Yamaguchi, S. Asai, and M. Mock, "A numerical model of avalanche breakdown in MOSFETs," IEEE Trans. Electron Devices, vol-ED-25, pp.825-832, 1978.
- (8) S. Selberherr, A. Shutz, and H.W. Potzl, "MINIMOS - a two-dimensional MOS transistor analyzer," IEEE Trans. Electron Devices, vol. ED-27, pp.1540-1550, 1980.
- (9) A. Husain and S.G. Chamberlain, "Three-dimensional simulation of VLSI MOSFET's : The three-dimensional simulation program WATMOS," IEEE Trans. Electron Devices, vol. ED-29, pp.631-638, 1982.
- (10) P.K. Ko, R.S.Muller, and C.Hu, "A unified model for hot-electron currents in MOSFET's," in IEDM Technical Digest, p.600, 1981.

- (11) S.Tam, F-C.Hsu, C.Hu, R.S.Muller and P.K.Ko, "Hot-electron currents in very short channel MOSFET's," IEEE Electron Device Letters, July, 1983.
- (12) K. Narita and K. Yamaguchi, "IGFET hot electron emission model," Solid-State Electronics, vol.23, pp.721-725, 1980.
- (13) C. Hu, "Lucky-electron model of hot electron emission," IEDM Technical Digest, p.22, 1979.
- (14) S. Tam, P. Ko, C. Hu, and R.S. Muller, "Correlation between substrate and gate currents in MOSFETs," IEEE Trans. Electron Devices, vol. ED-29, pp.1740-1744, Nov. 1982.
- (15) W. Shockley, "Problems related to p-n junctions in silicon," Solid-State Electronics, vol.2, pp.35-67, 1961.
- (16) J.F. Verwey, R.P. Kramer, and B.J. de Maagt, "Mean free path of hot electrons at the surface of boron-doped silicon," J. Applied Physics, vol.46, pp.2612-2619, 1975.
- (17) T.H. Ning, C.M. Osburn, and H.N. Yu, "Emission probability of hot electrons from silicon into silicon dioxide," J. of Applied Physics, vol.48, pp.286-293, 1977.
- (18) C.N. Berglund and R.J. Powell, "Photoinjection into  $\text{SiO}_2$ . Electron scattering in the image force potential well," J. of Applied Physics, vol.42, pp.573-579, 1971.
- (19) D.R. Young, "Electron current injected into  $\text{SiO}_2$  from p-type Si depletion regions," J. of Applied Physics, vol.47, pp.2093-2102, 1976.
- (20) T.N. Nguyen and J.D. Plummer, "Physical mechanisms responsible for short channel effects in MOS devices," IEDM Technical Digest, pp.596-599, 1981.
- (21) C.R. Crowell and S.M. Sze, "Temperature dependence of avalanche multiplication in semiconductors," Applied Physics Letters, vol. 9, No. 6, p.242,

1966.

- (22) R.P.Troutman,"Silicon surface emission of hot electrons," Solid-State Electronics, vol.21, pp.233-239, 1973.
- (23) R. Van Overstraeten and H. De Man," Measurement of the ionization rates in diffused silicon p-n junctions," Solid-State Electronics, vol. 13, pp.583-608, 1970.
- (24) W.N. Grant," Electron and hole ionization rates in epitaxial silicon at high electric fields," Solid-State Electronics, vol.16, pp.1189-1203, 1973.
- (25) J.L. Moll and R. Van Overstraeten," Charge multiplication in silicon p-n junctions," Solid-State Electronics, vol.6, pp.147-157, 1963.
- (26) S.M. Sze,Physics of Semiconductor Devices , J. Wiley and Sons, 1969.
- (27) D.Long," Scattering of conduction electrons by lattice vibrations in silicon," Physical Review,vol.120, pp.2024-2032,1960.
- (28) C.Y.Duh and J.L.Moll," Temperature dependence of hot electron drift velocity in silicon at high electric field," Solid-State Electronics, vol.11, pp.917-932, 1968.
- (29) E.Takeda, H.Kume, T.Toyabe, and S.Asai,"Submicrometer MOSFET structure for minimizing hot-carrier generation," IEEE Trans. Electron Devices, vol.ED-29, pp.611-615, 1982.



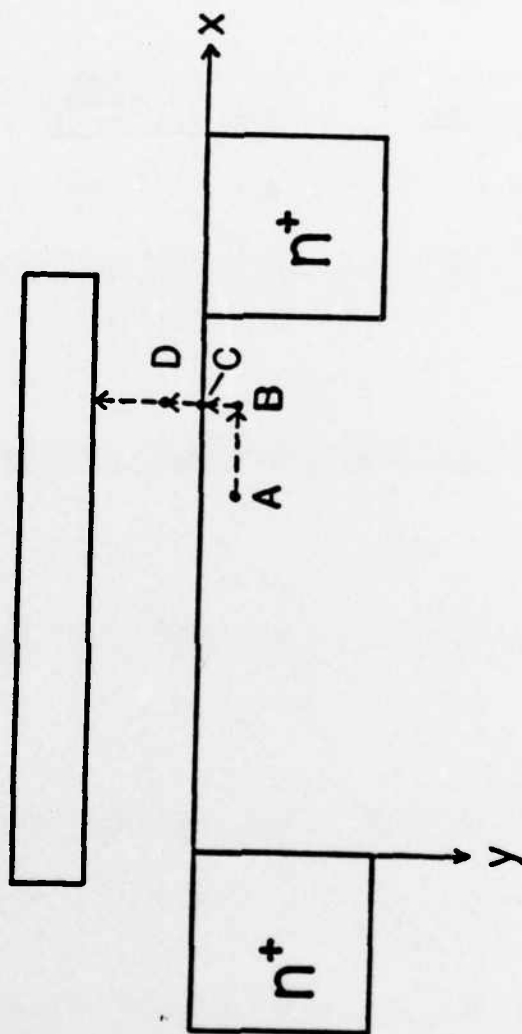
### Figure Captions

- (1) A cross-sectional view of the MOSFET. The three scattering probabilities in the model are illustrated.
- (2) A schematic illustration of the injected electron in the potential-distance space. The lucky electron travels a distance  $d$  to gain the energy needed to surmount the Si-SiO<sub>2</sub> potential barrier.
- (3) An illustration of the electron concentration versus the depth  $y$  at position  $x$  in the channel.
- (4) Measured source current ( $I_s$ ), substrate current ( $I_{sub}$ ), and gate current ( $I_{gate}$ ) versus the gate-to-source voltage ( $V_{gs}$ ) for a 1.3 $\mu m$  device from wafer A.
- (5) Measured  $I_{gate}$  versus the drain-to-source voltage ( $V_{ds}$ ) for the devices from wafer A (5a) and wafer B (5b). ( $V_{gs}=10V$  and  $V_{sub}=0V$ )
- (6) The field-limited case (6a) and the electrode-limited case (6c) for CHEI are illustrated. (6b) shows the condition where  $V_{ds} \approx V_{gs}$ . The point of maximum CHEI in each case is illustrated.
- (7) The dependence of  $I_{gate}$  on time illustrating the effect of charge trapping on CHEI gate current.
- (8) The normalized  $I_{gate}$  versus  $V_{ds}$  for the data shown in figure(5a). The solid (direct numerical integration of equation(10)) and the dash lines (analytical solution ie. equation(12)) represent theoretical results.
- (9) Constant normalized  $I_{gate}$  contour for the data shown in figures(5a&5b).
- (10) Experimental results on the correlation between the normalized  $I_{gate}$  and the normalized  $I_{sub}$  for constant  $V_{gs}$ . The solid lines represent theoretical results based on equations(14&15).

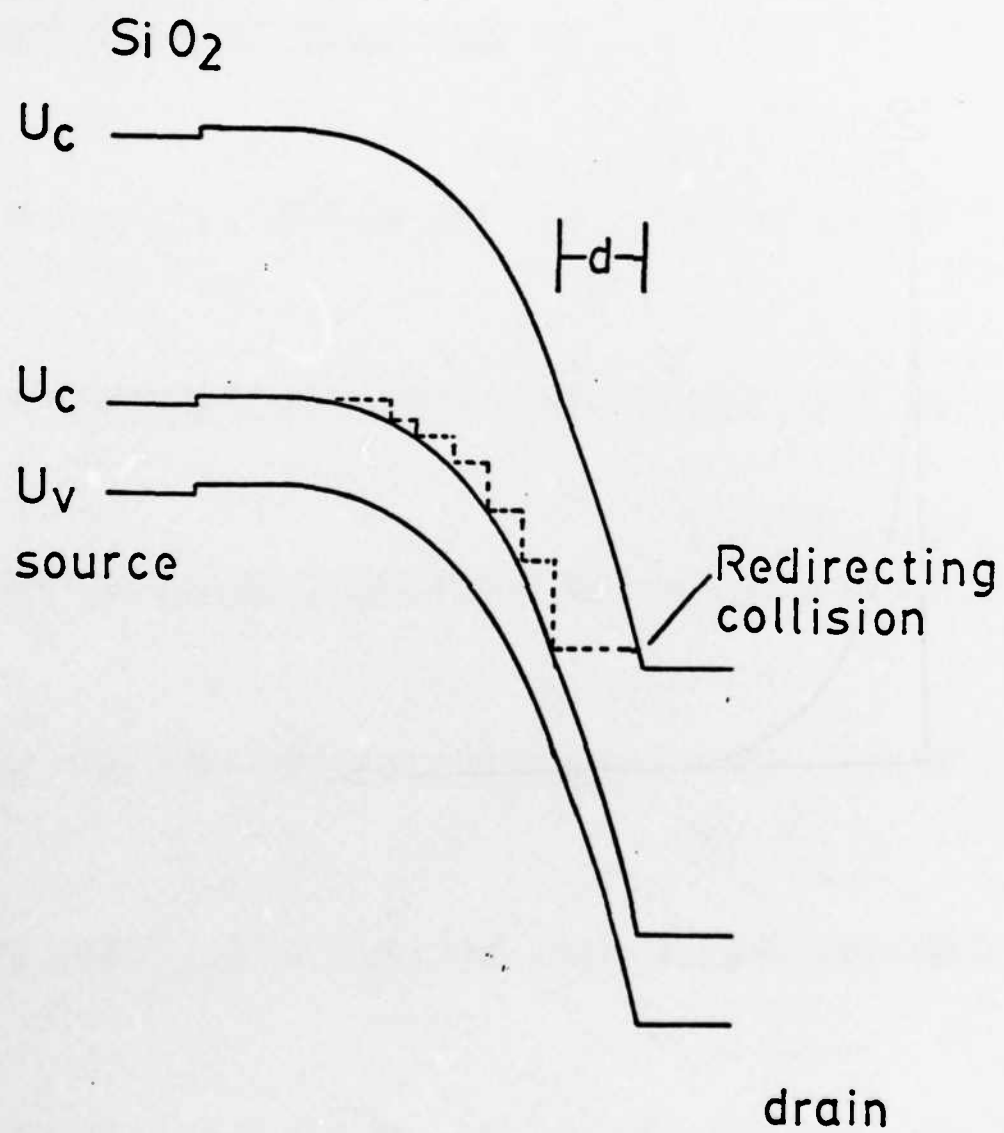
- (11) Dependence of the Si-SiO<sub>2</sub> potential barrier  $\phi_b$  and  $\frac{\phi_b}{E_i \lambda}$  on  $E_{ox}$  at the drain end. The data are derived from the experimental results shown in figure(10).
- (12) Comparison between experimental results and the calculated normalized  $I_{gate}$  using the correlation between the gate current and the substrate current.
- (13) Comparison between calculation and the data shown in figure(4). Numerical integration based on equation(10) is used to obtain the results on  $I_{gate}$ . Equation(13) is used to obtain the results on  $I_{sub}$ .
- (14) The temperature dependence of the normalized  $I_{gate}$  and the normalized  $I_{sub}$ .
- (15) Data shown in figure(14) is used to obtain the ratio of the normalized  $I_{gate}$  on temperature. Data obtained from Matsumoto et al. [4] is also plotted.

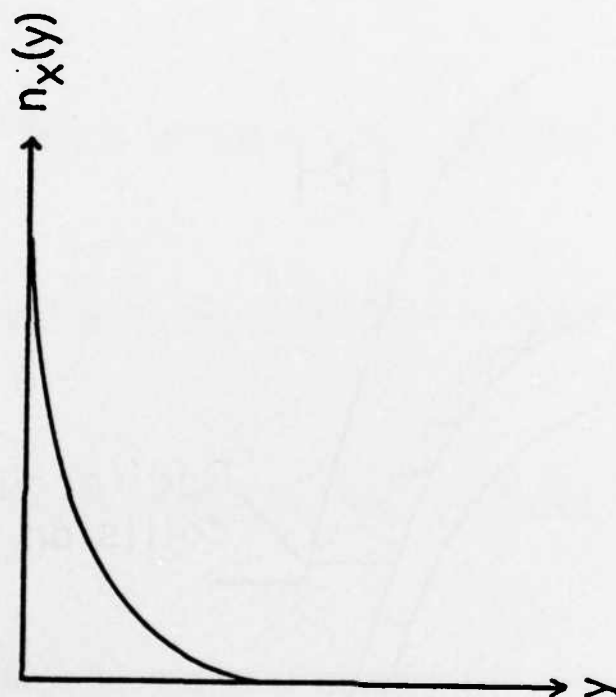
TABLE (1)

<u>wafer</u>	<u>X<sub>ox</sub> (nm)</u>	<u>N<sub>sub</sub> (10<sup>15</sup> cm<sup>-3</sup>)</u>	<u>X<sub>i</sub> (μm)</u>
A	82.1	6.7	0.35
B	35.8	6.7	0.30

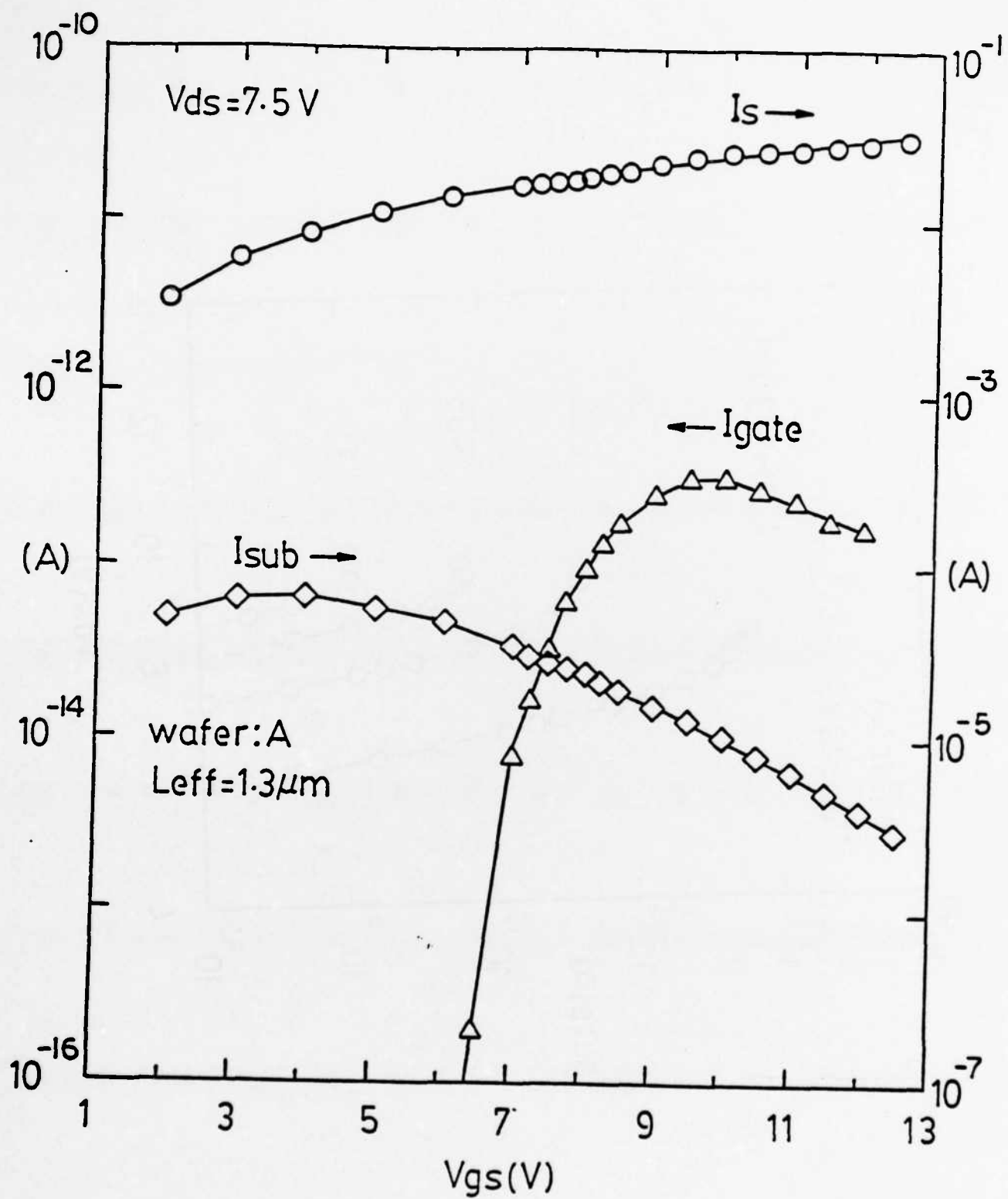


(1)

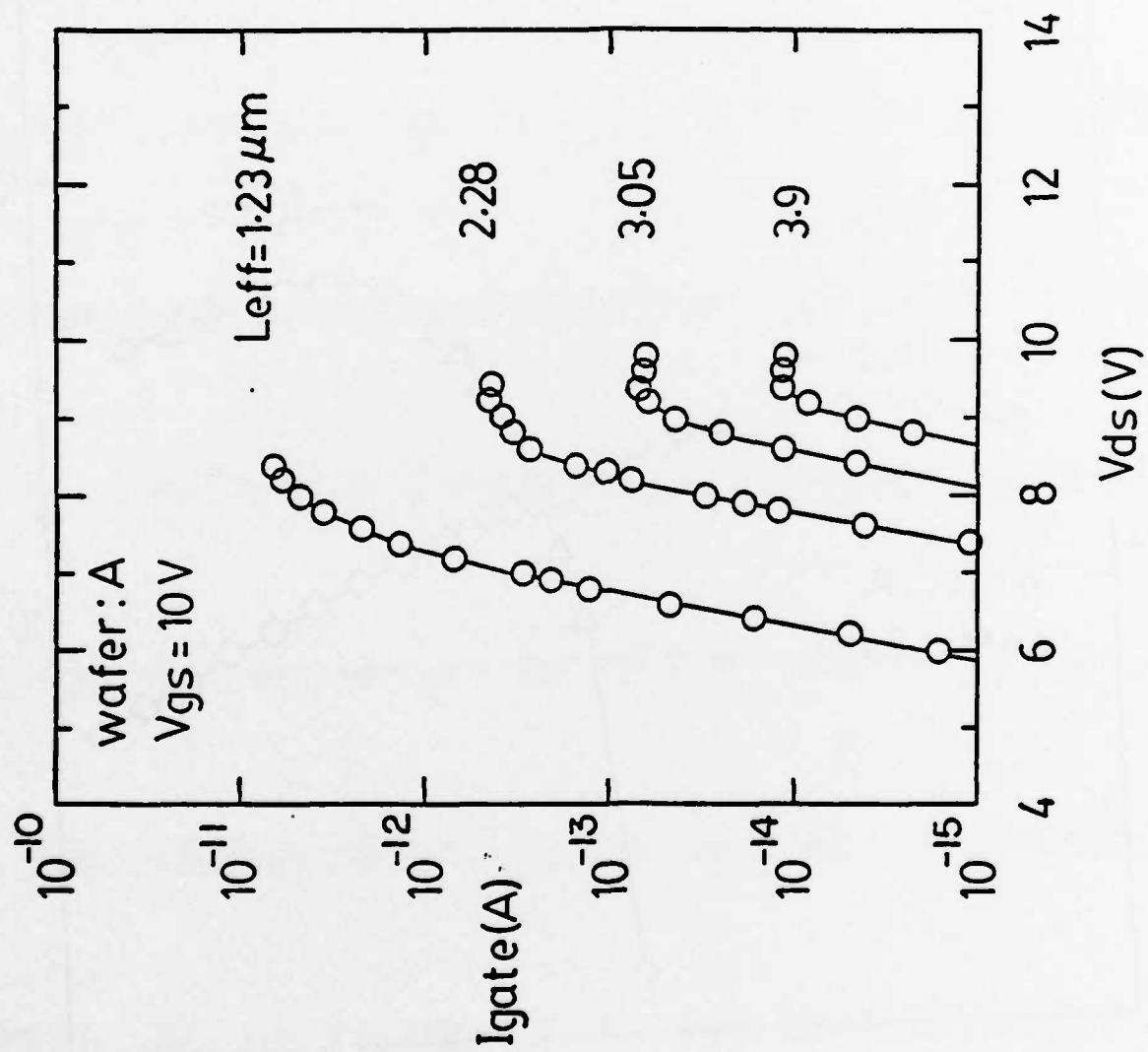




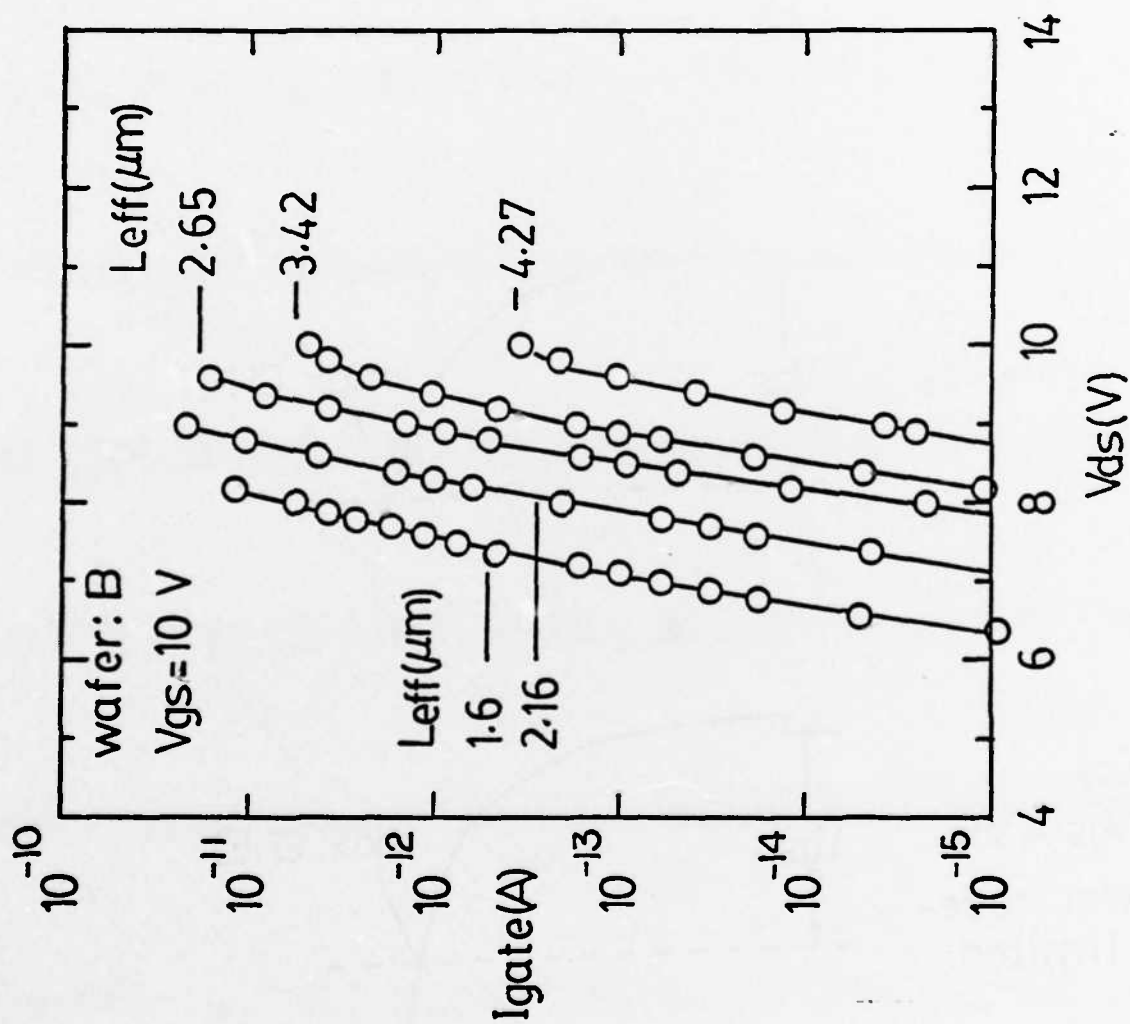
(3)







(5a)

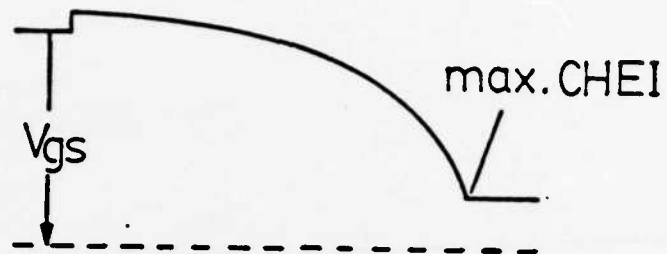


(5b)

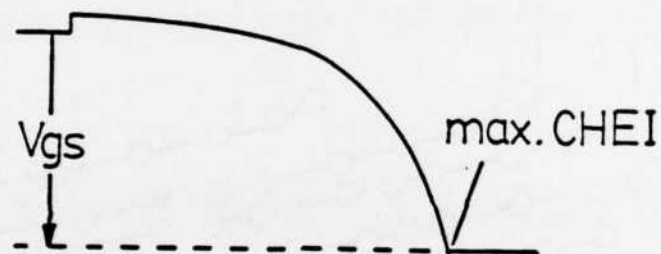
source

drain

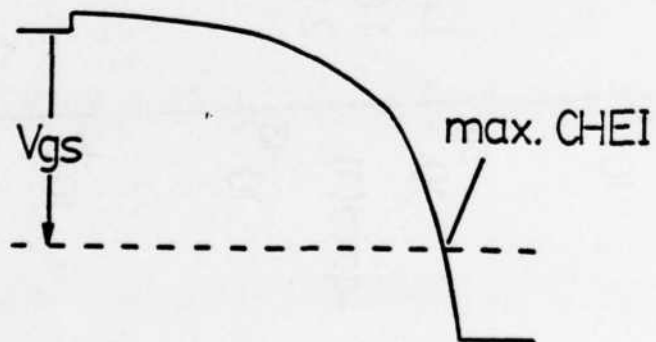
(a)  
 $V_{gs} > V_{ds}$   
field-limited

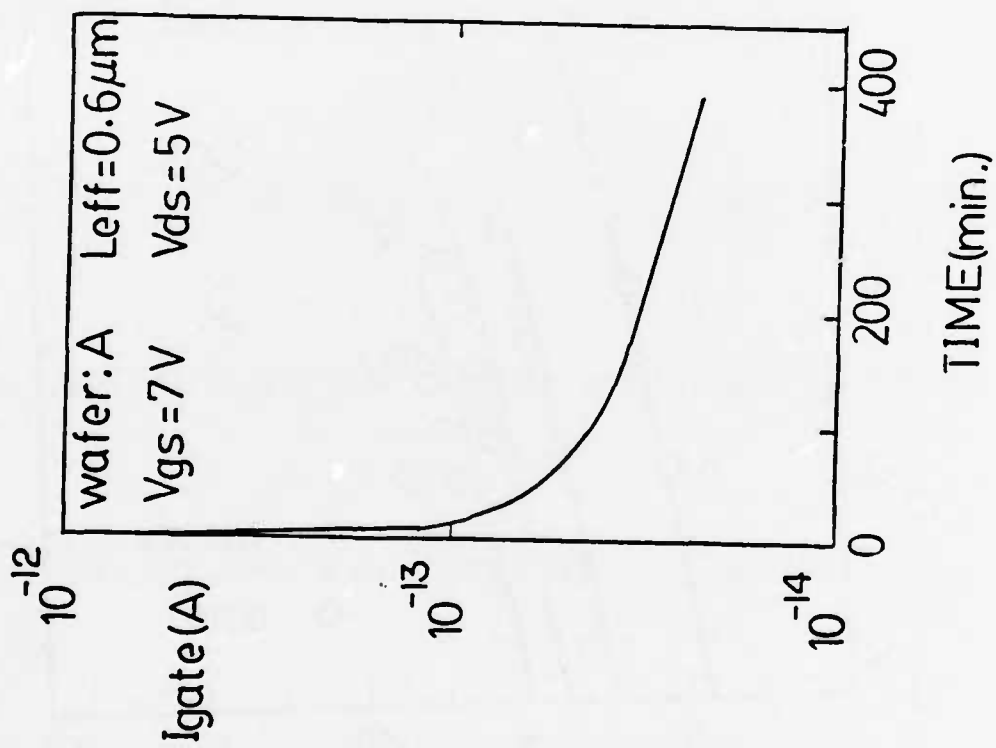


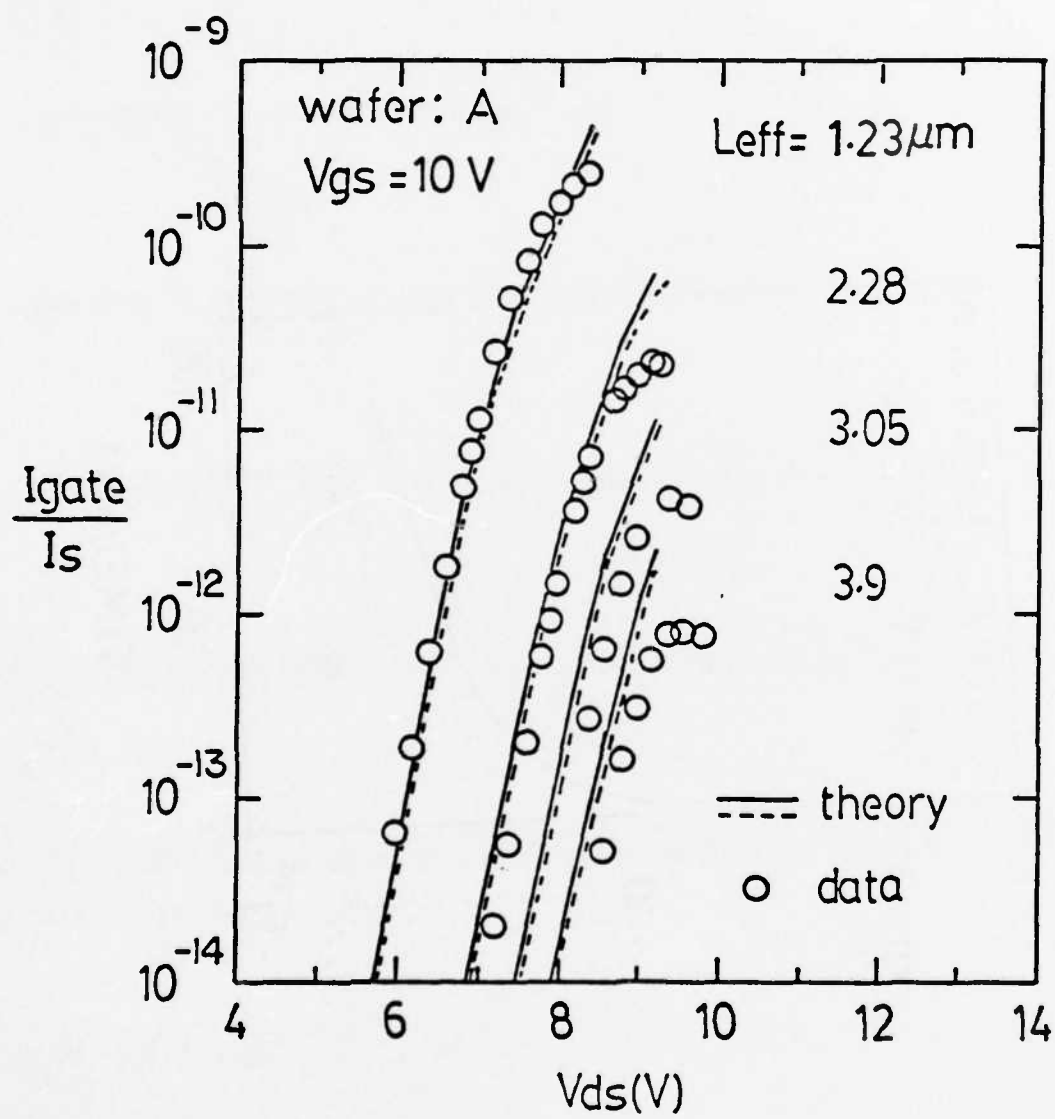
(b)  
 $V_{gs} \approx V_{ds}$

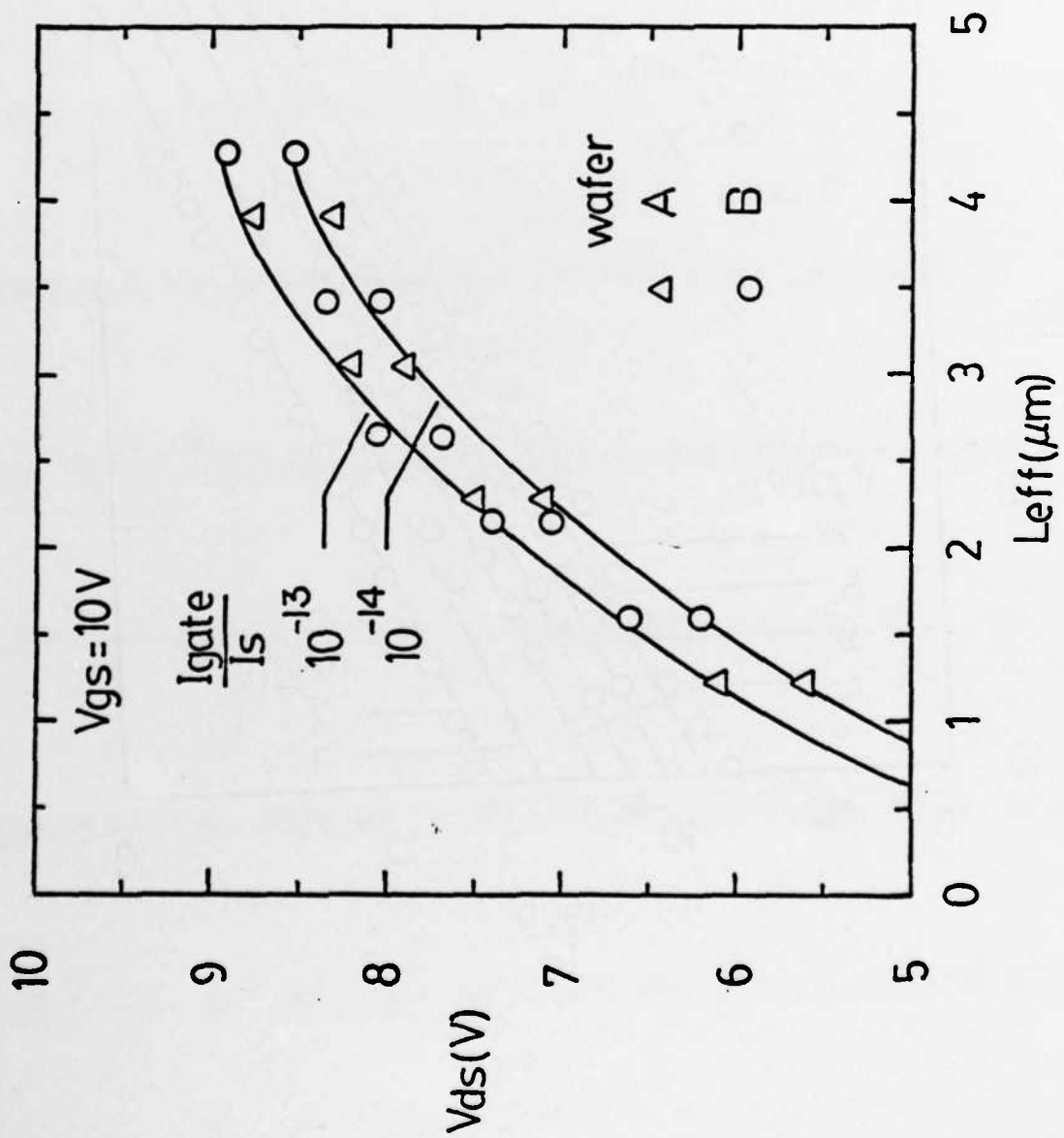


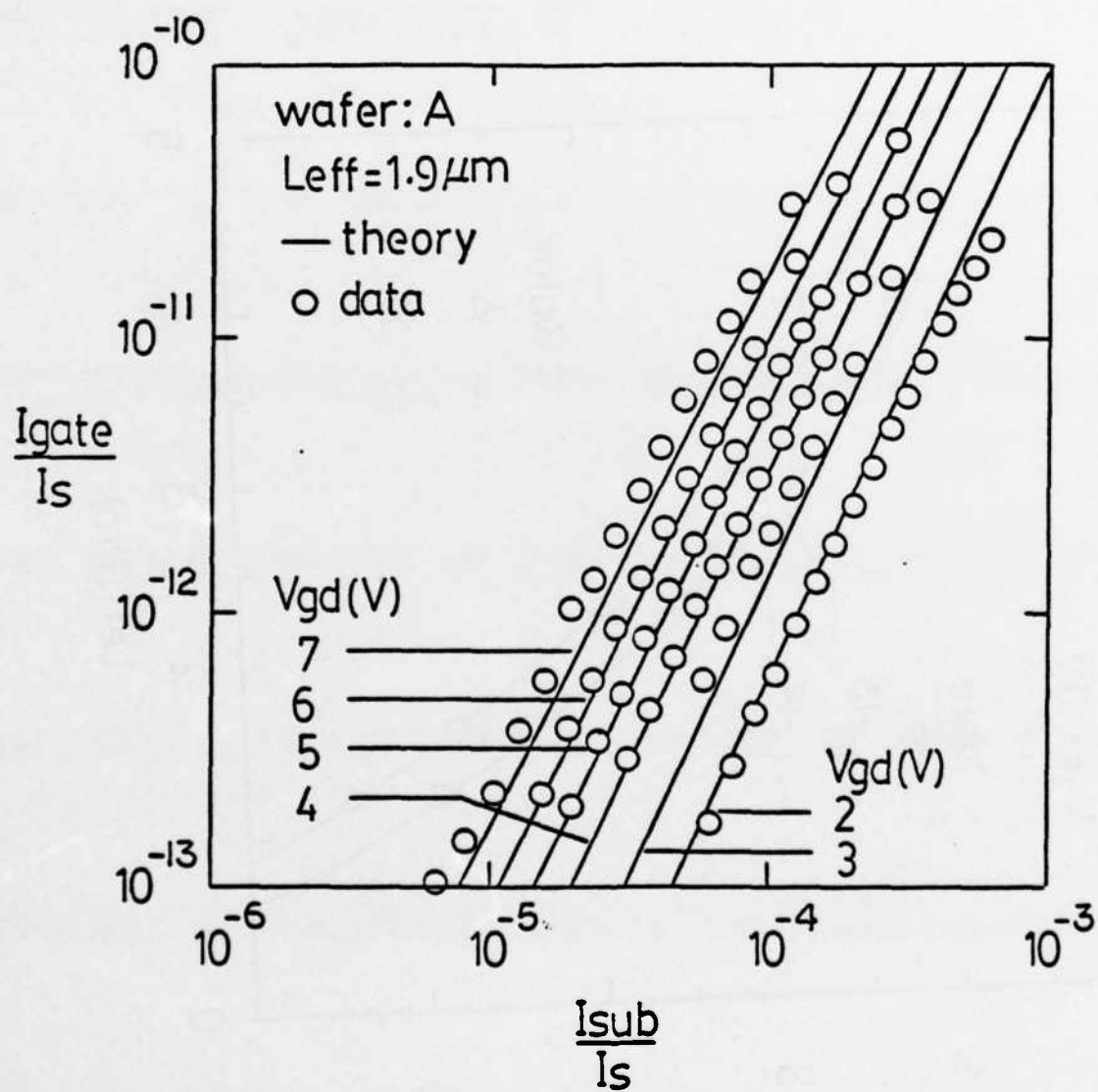
(c)  
 $V_{gs} < V_{ds}$   
electrode-limited



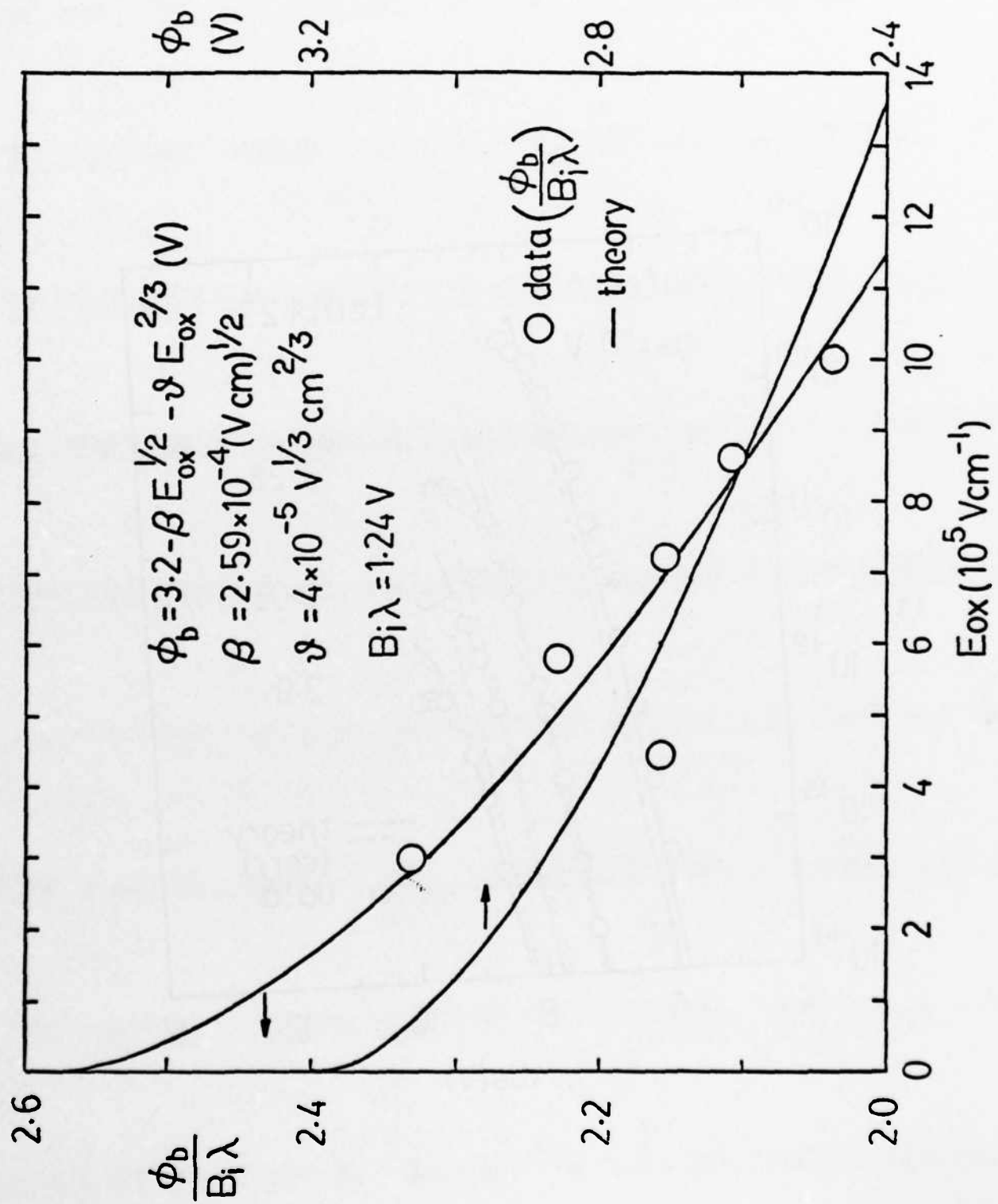


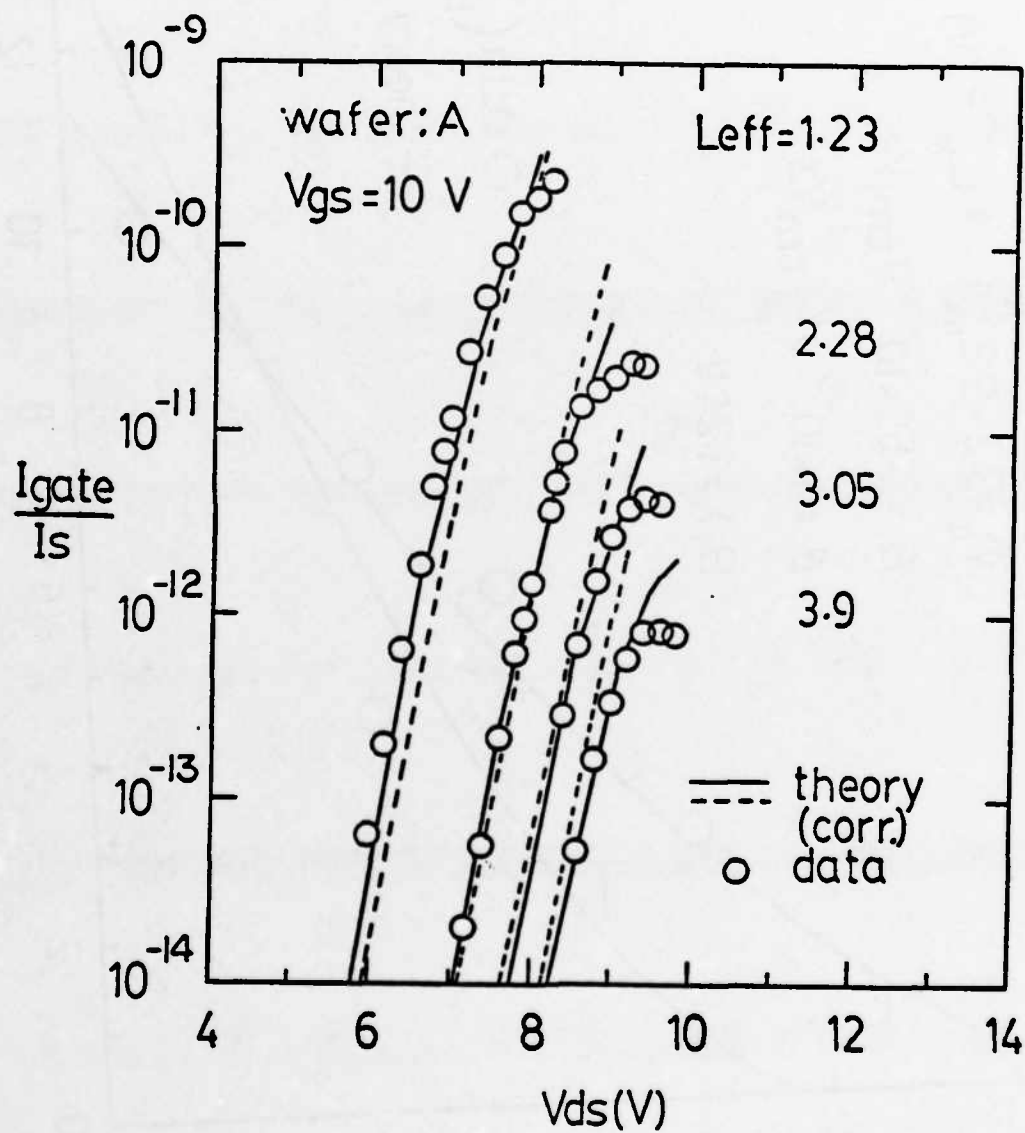




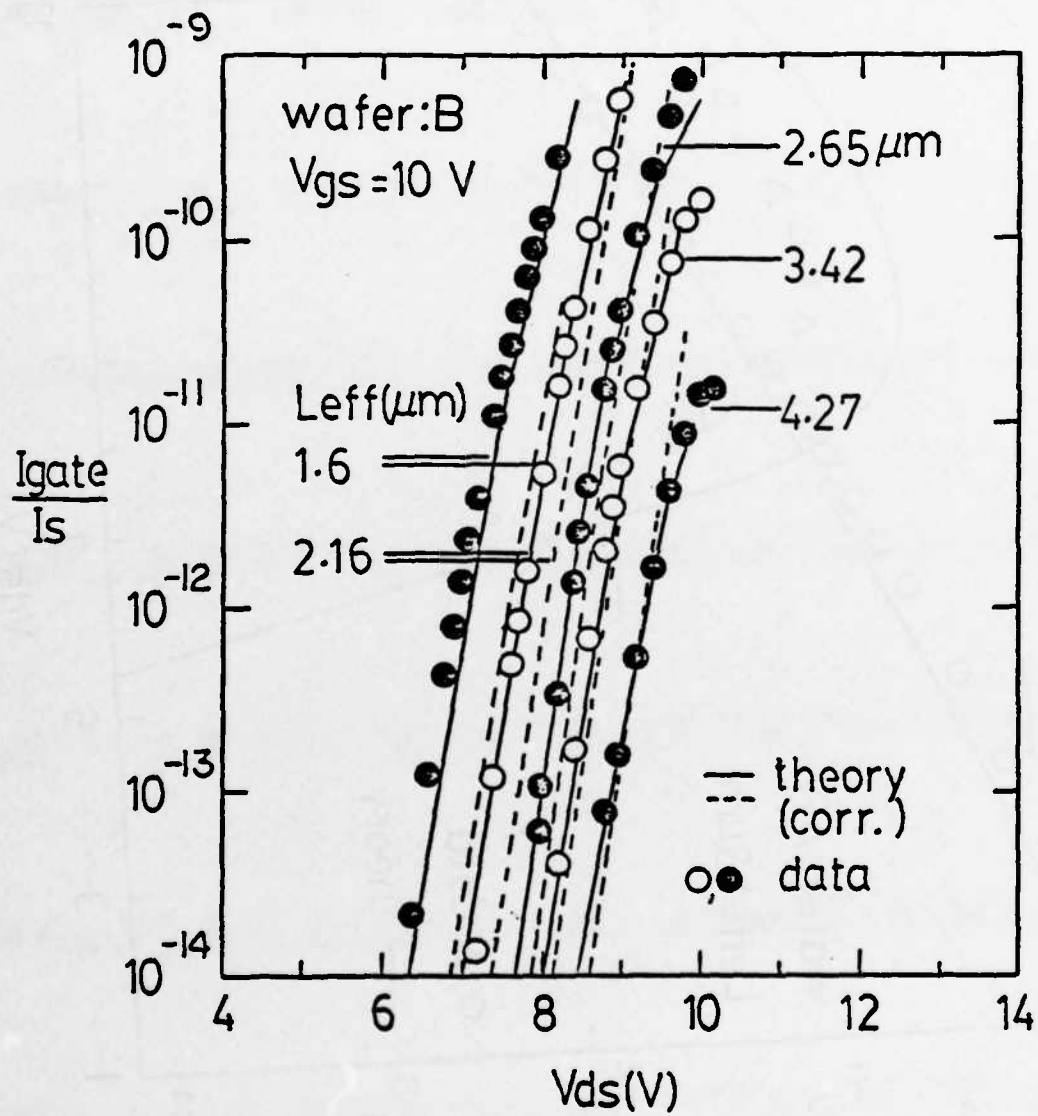




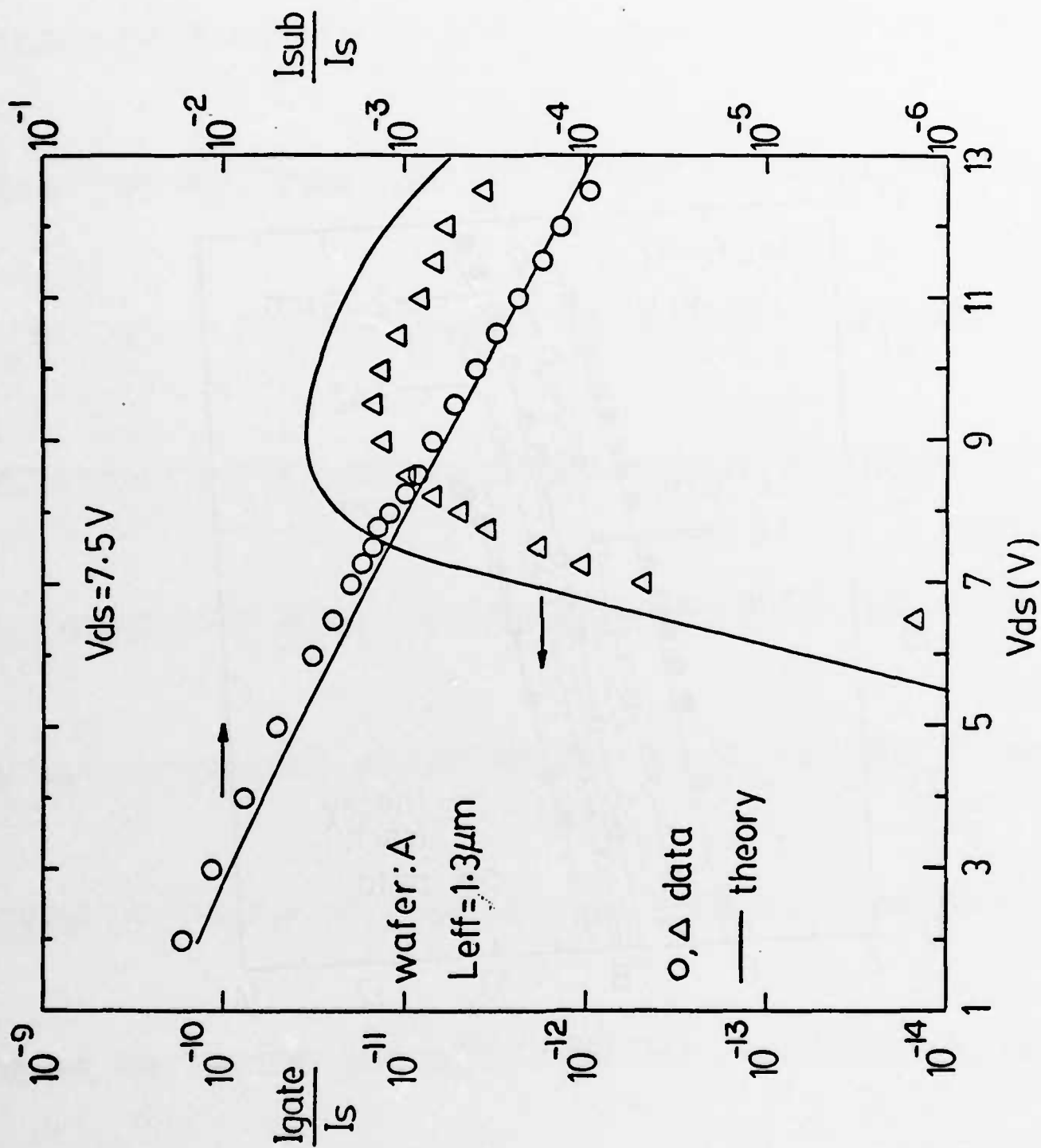


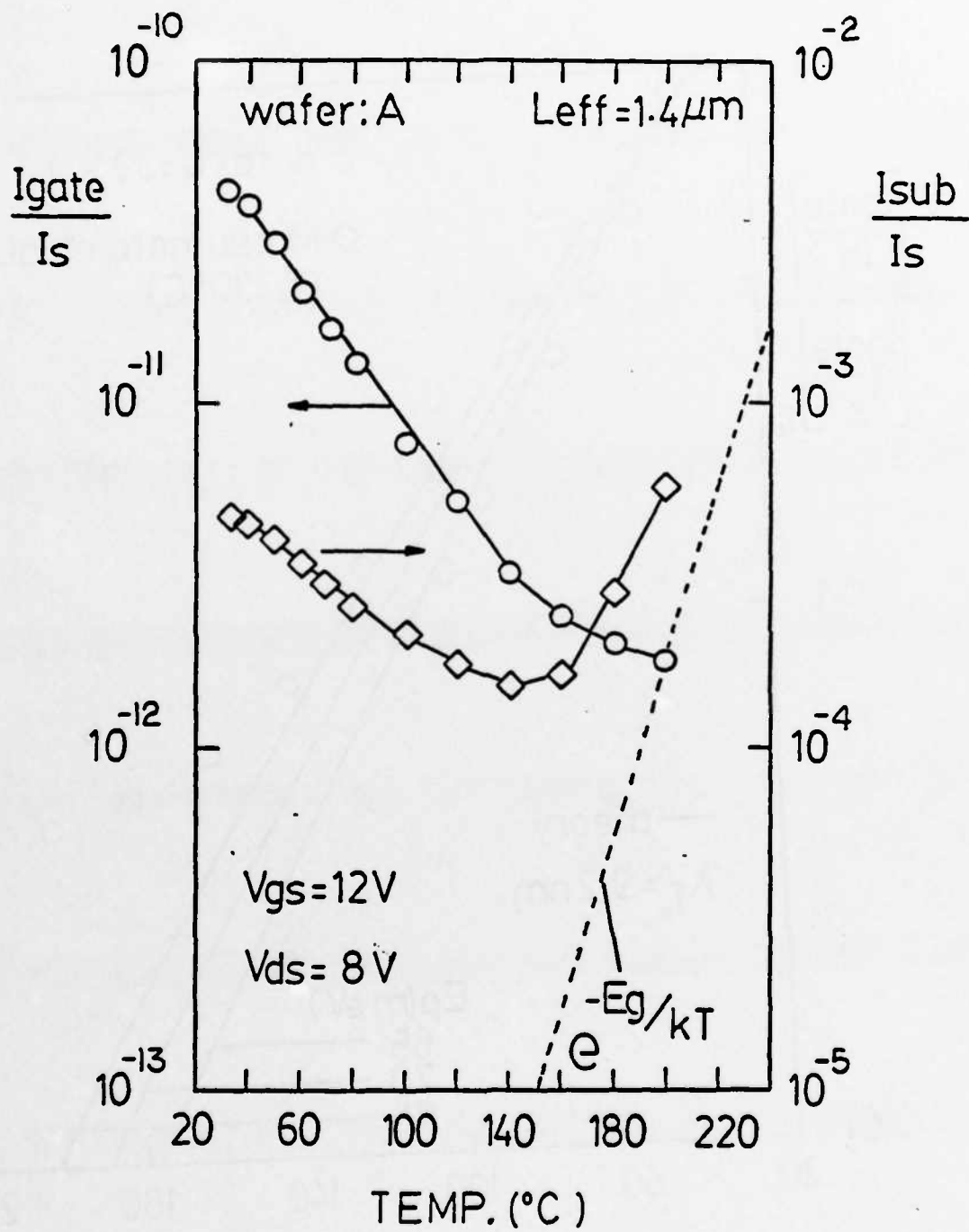


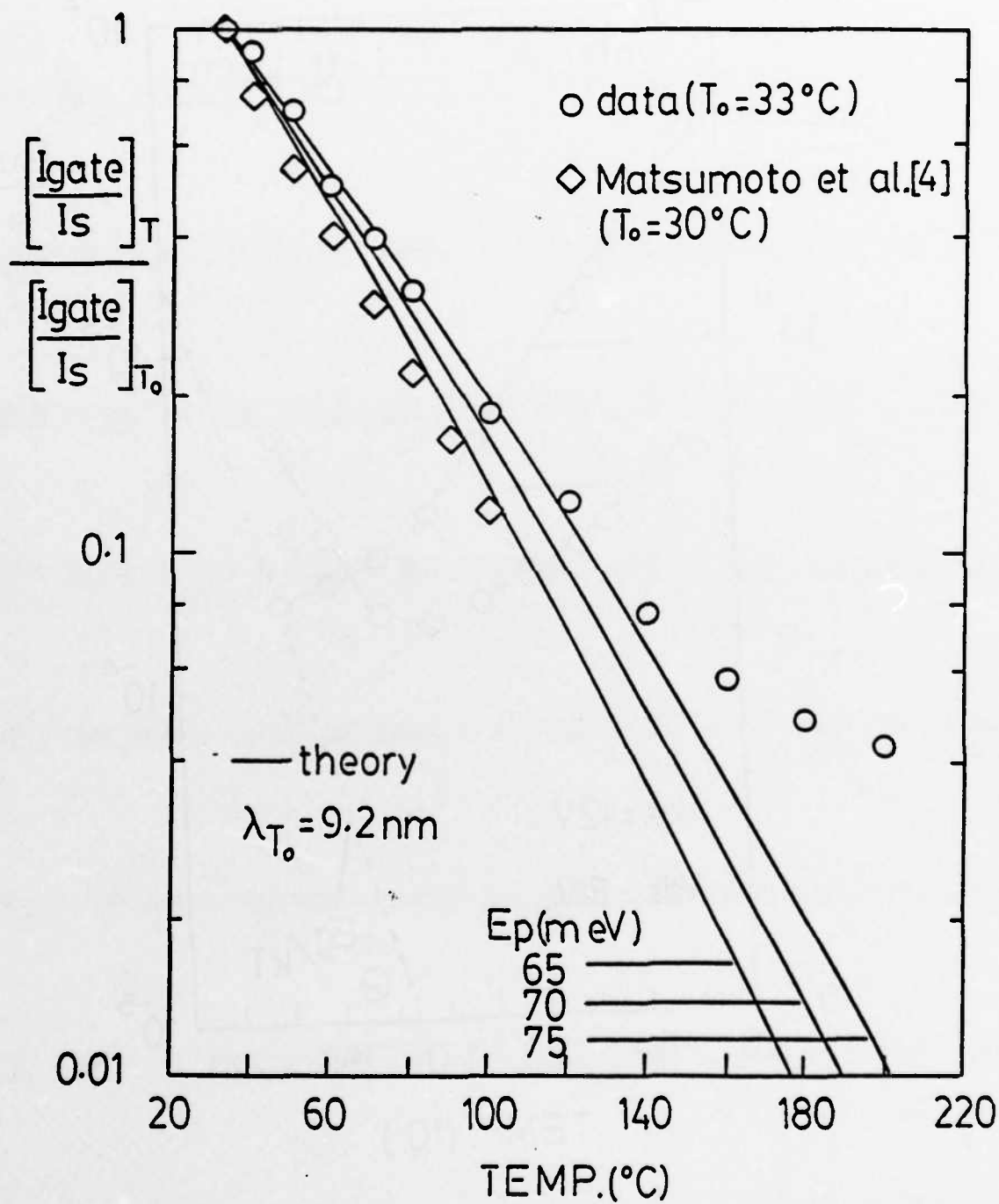
(12a)



(126)







## Hot Carriers Induced Degradation in Thin Gate Oxide MOSFETs

M-S. Liang, C. Chang, W. Yang, C. Hu and R. W. Brodersen

Department of Electrical Engineering and Computer Sciences  
Electronic Research Laboratory  
University of California, Berkeley

### Abstract

The degradation of thin gate oxide ( $\sim 100\text{\AA}$ ) n and p-channel MOSFETs subjected to the substrate hot carrier injection is discussed. The generation of oxide trapped charges is observed to be sub-linearly dependent on the applied oxide field, while the generation of interface trapped charges shows a linear dependence on the applied oxide field. The generation rates are found to be a function of carrier fluence and the oxide field, and are independent of the injection current density. The generation of interface traps correlates well with the mobility and subthreshold current degradation. An oxide field around  $5\text{MV/cm}$  is found to be a critical value for accelerating device degradation. There is no significant interface trap generation under substrate hot hole injection for the hole fluence up to  $2 \times 10^{17}/\text{cm}^2$ . The threshold voltage shifts decrease with increasing applied substrate bias. Possible mechanisms are discussed to account for the experimental data.

### 1. Introduction

Scaled devices for NMOS and CMOS integrated-circuits have become increasingly important. As the channel and oxide fields increase, hot-carrier emission imposes serious limitations on the long-term reliability of scaled VLSI circuits. Studies of device degradation to date have been mostly based on channel hot-carrier injection, which is very localized in nature. Consequently, the results have been difficult to interpret and generalize. It is almost impossible, for example, to study the degradation mechanisms with this injection technique. Substrate hot carriers can be uniformly injected throughout the channel [1, 2]. However, this technique has not been used to study the degradation of transconductance, channel mobility, subthreshold current, etc. This paper reports the device degradations due to the substrate hot-carrier injection into thin gate oxide ( $\sim 100\text{\AA}$ ) p and n channel MOSFETs.

### 2. Device Preparation

The n-channel and p-channel transistors used in our investigation are fabricated with the standard silicon gate process. The starting material are all  $\langle 100 \rangle$  substrates with substrate resistivities in the range of  $10\text{--}25\text{ ohm-cm}$ . The gate oxide ( $\sim 100\text{\AA}$ ) is grown in dry oxygen ambient for  $\sim 16$  min. and followed by a 3-min. nitrogen anneal at the same temperature. The polysilicon gate is arsenic doped to avoid dopant diffusion through the thin oxide. The channel region is ion

implanted with a final surface concentration of  $\sim 10^{17}/\text{cm}^3$ . The drain and source are implanted with arsenic for the n-channel devices to give a final junction depth of  $\sim 0.7\text{ }\mu\text{m}$  and with boron for the p-channel devices to give a final junction depth of  $\sim 1.2\text{ }\mu\text{m}$ . Special process steps are also taken to suppress any drain (source) to gate current leakage as a result of drain (source) ion implant damage on the thin oxide. The devices are all sintered in forming gas at  $400^\circ\text{C}$  for 15 min. after metallization.

The threshold voltages are  $0.5\text{V}$  and  $-1.4\text{V}$  for n and p-channel devices respectively. Subthreshold current slope is  $\sim 80\text{mV/decade}$ . Channel mobilities are  $450\text{ cm}^2/\text{V-sec}$  for electrons and  $170\text{ cm}^2/\text{V-sec}$  for holes. The drain (source) junction breakdown voltage is  $\sim 18\text{V}$  for  $n^+p$  junction and  $\sim 32\text{V}$  for  $p^+n$  junction with zero gate bias.

### 3. Experimental Details

The experimental configuration used for substrate hot-carrier injection is shown in Fig. 1. Gate bias ( $V_g$ ) and substrate bias ( $V_{sub}$ ) are separately controlled with source and drain grounded. The gate bias is always large enough to invert the surface so that the channel is near the ground potential. The minority carriers, electrons in n-channel devices and holes in p-channel devices, are injected from the adjacent forward-biased p-n junction which is located  $100\text{ }\mu\text{m}$  away. Most of the minority carriers that are injected will recombine in the substrate giving rise to the substrate current. However, a small fraction of these injected carriers will enter the deep depletion region and will be accelerated toward the surface inversion layer. Those arriving at the  $\text{Si-SiO}_2$  interface with sufficient energy to surmount the barrier will be injected in the  $\text{SiO}_2$ . The fluence (number of carriers/unit area) of injected carriers is monitored via the gate current measurement. High frequency (H-F) ( $1\text{MHz}$ ) C-V, quasi-static C-V, and Fowler-Nordheim (F-N) I-V characteristics are measured before and after carrier injection to determine the flat band voltage, the interface trap density and the bulk oxide charge density respectively. The drain current is measured with the gate voltage varying from  $1\text{V}$  below threshold up to  $5\text{V}$  at a constant drain voltage of  $50\text{mV}$ . The transistor threshold voltage and transconductance were extracted from the measured drain current data using the current to voltage relationship in the linear region.



## 4. Results and Discussions

### 4.1. Substrate Hot-Electron Effect

The effect of hot carrier fluence and applied oxide field on the charge trapping and therefore  $\Delta V_T$  can be obtained by operating a device at the conditions A, B, C and D shown in Fig. 2. A, B and C have approximately the same gate current, but their oxide fields are 4, 8 and 10 MV/cm respectively. D has the same oxide field (8 MV/cm) as B except that the gate current is 5 orders of magnitude lower. For the same stressing time (1 hr.), there is almost no observable threshold voltage shift ( $\Delta V_T$ ) in the device operated under condition D. However,  $\Delta V_T$  gradually increases from condition A to condition C. The above observation indicates that the charge trapping effect is a function of both carrier fluence and oxide field.

Threshold voltage shift versus electron fluence for various gate bias conditions is shown in Fig. 3.  $\Delta V_T$  increases monotonically with increasing fluence and oxide field ( $E_{ox} > 5$  MV/cm).  $\Delta V_T$  is approximately the same and tends to saturate for oxide field less than 5 MV/cm. However, a linear increase in the trapped charge density due to trap filling and a constant trap generation rate has been observed under high oxide field and large fluence [4]. The slopes in the high fluence region, which are proportional to the trap generation rate, are shown in the inset of Fig. 3. The gate current under the same substrate bias and oxide field can be controlled by varying the injecting current. It has been found that  $\Delta V_T$  is a very weak function of  $I_g$  over the range 0.1 nA to 1 nA, i.e.,  $\Delta V_T$  is solely dependent on the amount of electron fluence through the oxide at this bias. The initial negative value of  $\Delta V_T$  for high gate biases, corresponding to positive charge trapping, has been tentatively identified as hole trapping near the gate interface. The trapped holes may come from the electron-hole pair generation in the polysilicon gate due to high energy electron injected from the substrate or from impact ionization in  $\text{SiO}_2$ . The trapped holes can be de-trapped or neutralized by reverse gate bias and retrapped by hot hole filling.

The threshold voltage shift is caused by the oxide trapped charge and the interface trapped charge. The effects of these two factors can be separated via comparison of the positive gate Fowler-Nordheim  $I_g$ - $V_g$  shift ( $\Delta V_g^+$ ) and  $\Delta V_T$ . That is, for first order consideration,  $\Delta V_g^+$  is insensitive to the interface-trapped charge. The difference between  $\Delta V_T$  and  $\Delta V_g^+$ , therefore, will be the contribution from the interface trapped charge ( $\Delta V_{ox}$ ) [5]. Fig. 4 is a plot of the voltage changes at a constant carrier fluence versus the applied oxide voltage.  $\Delta V_g^+$ , corresponding to the oxide trapped charge, decreases at high gate biases because of the significant hole trapping effect. However, the generation of interface trapped charge shows a linear dependence on the applied oxide field. An oxide field around 5 MV/cm is found to be the critical value for accelerating device degradation. The generated interface-trap distribution, deduced from the quasi-static C-V measurement, is similar to the interface traps generated by F-N stressing [5], radiation ionization [6], and internal photoemission [7]. Namely, a characteristic peak centered around 0.65 eV

above the valence band edge is observed. The interface trap density gradually decreases for energies below the peak. This peak state is largely responsible for the mobility and subthreshold degradation. The interface-trap generation is much more significant in the case of substrate hot electron injection when compared with F-N injection where the carrier is essentially in thermal equilibrium at the interface. The generation of interface traps and fixed charges close to the Si-SiO<sub>2</sub> interface ( $\Delta V_{ox}$  in Fig. 4) correlates well with the mobility and subthreshold current degradation which is shown in Fig. 5.

### 4.2. Substrate Hot-Hole Effect

The p-channel transistor is subjected to stressing similar to the n-channel case. However, in this case the hot carriers are holes, and higher substrate bias and injection current are necessary due to the larger barrier height. The substrate hot hole current from the substrate depletion region and F-N electron tunneling current from the gate are shown in Fig. 6. The steeper slope of the substrate hot hole current versus gate bias comparing with substrate hot electron current (shown in Fig. 2) is probably due to the large number of scattering events within the barrier lowering region because of the extremely low mobility of holes in  $\text{SiO}_2$ . The stressing conditions are chosen with the gate bias less than 8V to avoid significant electron tunneling. Hole trapping can be detected from the transistor negative  $\Delta V_T$  as well as negative F-N I-V shifts. It has been found that the subthreshold slope suffers no observable degradation up to  $2 \times 10^{17}/\text{cm}^2$  of injected hot holes. By measuring the same transistor threshold voltage shift with the H-F C-V flatband voltage shift, one can conclude no significant interface trap generation above the midgap. This conclusion is then doubly checked by the quasi-static C-V measurement which showed very low interface-trap density after hot hole injection. This indicates no significant interface-trap generation in the band gap.

### 4.3. Substrate Bias Effect

The threshold voltage shift is found to be a strong function of substrate bias, which is shown in Fig. 7. Namely, less degradation is observed at higher substrate bias, and it tends to saturate at lower substrate bias. The subthreshold current slope ( $\Delta S$ ), however, is shown to be a weak function of  $V_{sub}$ . Both  $\Delta V_T$  and  $\Delta S$  are strong function of applied gate bias in the substrate hot-electron injection case.

Three possible mechanisms have been considered to account for the experimentally observed substrate bias dependence: (1) impact ionization in the silicon depletion region, followed by subsequent hot carrier injection over the SiO<sub>2</sub>-Si barrier, (2) impact ionization in SiO<sub>2</sub> and subsequent trapping of carriers and (3) carrier energy dependence of trap capture cross section. While in general all three processes may take place in the above experiments, some processes may dominate over the others, depending on experimental conditions.

For the p-channel device shown in Fig. 7, impact ionization caused by injected holes can occur in the silicon depletion region due to the

large substrate bias. Some of the impact ionized hot electrons will have sufficient energy to overcome the oxide barrier and get injected into  $\text{SiO}_2$ , despite the negative gate voltage [8]. A portion of the injected electrons can be captured by the trapped holes with relatively high probability, due to the large capture cross sections of the coulombic traps. This results in a charge compensation effect, which leads to a reduction of  $\Delta V_T$ . Since the degree of impact ionization increases with the substrate bias, the end result is the observed decrease in  $\Delta V_T$  with increasing substrate bias.

For the n-channel device, although impact ionization in the silicon depletion region can also occur, its degree is substantially lower due to the much lower substrate bias. In addition, hole injection into  $\text{SiO}_2$  in this case is much less likely [8], probably due to the higher barrier for the hole injection. However, for the n-channel device at high oxide field ( $V_g=8\text{V}$ , corresponding to  $\sim 8\text{MV/cm}$ ), impact ionization in the  $\text{SiO}_2$  layer may take place due to hot electrons emitted from the substrate, which leads to net hole trapping [10], causing a reduction in  $\Delta V_T$ . This effect is more pronounced for higher substrate biases, since the average energy of the hot electrons entering  $\text{SiO}_2$  is higher, which is consistent with the observed substrate bias effect.

For n-channel devices with low oxide field ( $V_g=3\text{V}$ , corresponding to  $\sim 3\text{MV/cm}$ ), impact ionization in  $\text{SiO}_2$  is much less likely. In this case, the carrier energy dependence of the oxide trap capture cross section may play a dominant role. As the energy of the injected electron increases, the capture probability decreases [11], which can explain the substrate bias dependence observed here. An alternative interpretation is that, as the energy of the injected electrons increases, the centroid of the trapped electrons will move away from the  $\text{SiO}_2/\text{Si}$  interface, which will be reflected in the external measurement as a smaller  $\Delta V_T$ .

## 5. Other Discussions

(1) Threshold voltage shift as a function of substrate bias has been studied by using nonavalanche injection technique [1]. At first glance, it appears that the results shown here are contradictory to those reported in Ref. [1] which showed that  $\Delta V_T$  increases with increasing  $V_{\text{sub}}$ . The discrepancy arises because a fixed stressing time was used in [1] while a fixed carrier fluence is used in this study. We believe carrier fluence is a more fundamental parameter, which offers more useful comparisons for the studies of device degradation.

(2) The interface trap generation caused by substrate hot-hole injection in p-channel transistors is less than that caused by the substrate hot-electron injection in n-channel transistors for the same carrier fluence. This is different from the case of avalanche hole injection [12].

(3) For large  $V_{\text{sub}}$ , it appears that the  $\Delta V_T$  degradation caused by substrate hot-hole injection is less than the degradation caused by substrate hot-electron injection. For low  $V_{\text{sub}}$ , however, the substrate hot-hole injection causes much more significant  $\Delta V_T$  degradation. This is shown in Fig. 8.

(4) Due to the strong correlation of  $\Delta V_T$  with  $V_{\text{sub}}$  and  $V_g$  as shown in Fig. 7, it is possible to choose an optimal set of operating parameters for  $V_{\text{sub}}$  and  $V_g$  to achieve the lowest  $\Delta V_T$  and interface trap generation.

## 6. Acknowledgement

The authors wish to thank Prof. T.P. Ma of Yale University for useful discussions. Research was sponsored by DARPA under grant N00039-81-K-0251.

## References

- [1] J.F. Verwey, J. Appl. Phys., 44, 2681 (1973).
- [2] T.H. Ning, J. Appl. Phys., 47, 3203 (1976).
- [3] B. Eitan, D. Frohman-Bentchkowsky, J. Shapir, and M. Balog, Appl. Phys. Lett., 40, 523 (1982).
- [4] M-S. Liang and C. Hu, 1981 IEDM Tech. Digest, p. 385.
- [5] M-S. Liang, Y.T. Yeow, C. Chang, C. Hu, and R.W. Brodersen, 1982 IEDM Tech. Digest, p. 50.
- [6] T.P. Ma, G. Scoggan, and R. Leone, Appl. Phys. Lett., 27, 61 (1975).
- [7] V. Zekeriya and T.P. Ma, Appl. Phys. Lett., 43, 95 (1983).
- [8] K.K. Ng and G.W. Taylor, IEEE Trans. Electron Dev., ED-30, 871 (1983).
- [9] D.J. DiMaria, Proc. Intern'l Topical Conf. on Physics of  $\text{SiO}_2$  and its interfaces, Yorktown Heights, New York, 1978.
- [10] P. Solomon and N. Klein, Solid-State Comm., 17, 1397 (1975).
- [11] A.G. Milnes, "Deep Impurities in Semiconductors", Wiley Interscience, NY, 1973, Chap. 5.
- [12] E. Takeda, Y. Nakagome, H. Kume, N. Suzuki, and S. Asai, IEEE Trans. Electron Dev., ED-30, 675 (1983).

Fig. 1 Schematic diagram of substrate hot-carrier injection.

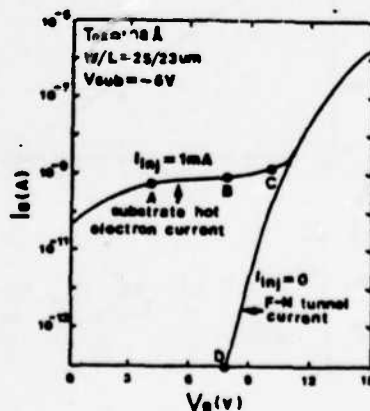
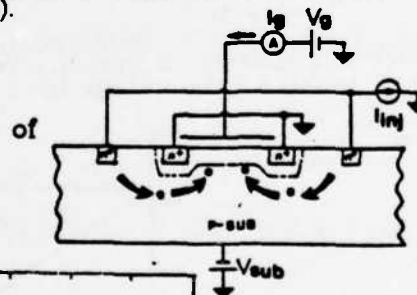


Fig. 2  $I_{\text{inj}}$  vs.  $V_g$  for F-N tunneling (with  $V_{\text{sub}}=0$ ,  $I_{\text{inj}}=0$ ) and substrate hot electron injection (with  $V_{\text{sub}}=-5\text{V}$ ,  $I_{\text{inj}}=1\text{mA}$ ).

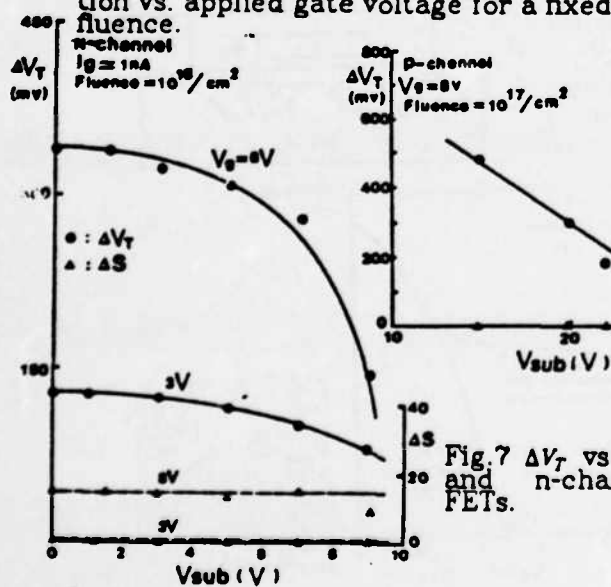
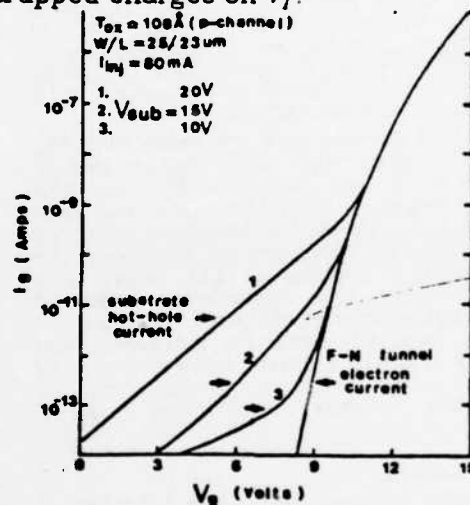
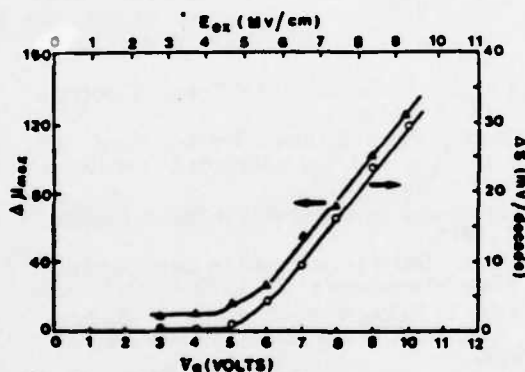
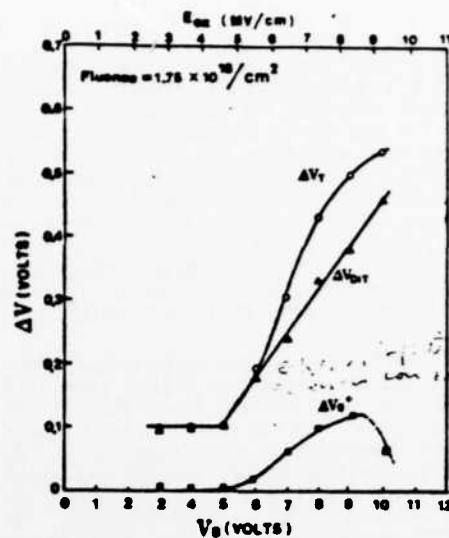
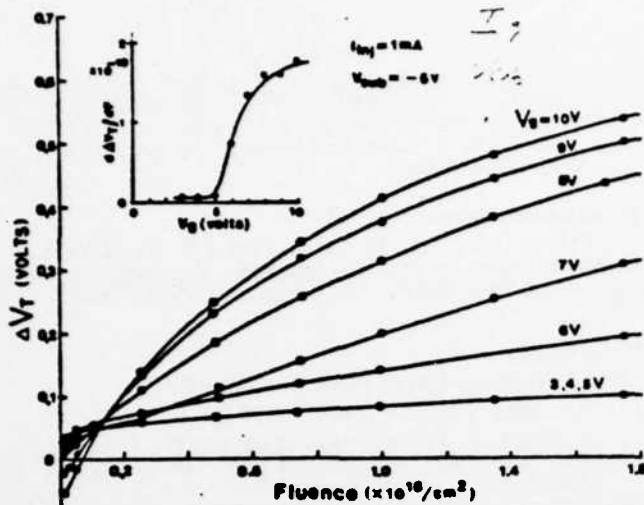
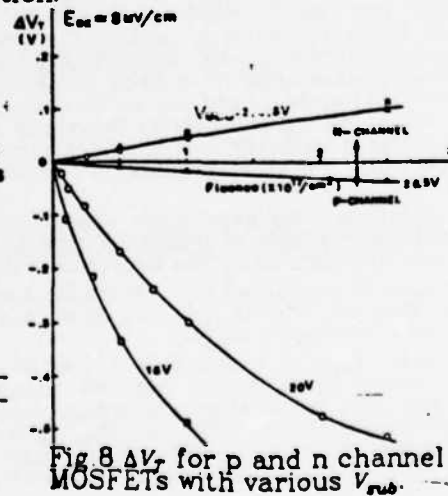


Fig. 6  $I_g$  vs.  $V_g$  for substrate hot-hole injection.



## Carrier Tunneling Related Phenomena In Thin-Oxide MOSFET's

C. Chang, M-S Liang, C. Hu and R. W. Brodersen

Department of Electrical Engineering and Computer Sciences  
Electronic Research Laboratory  
University of California, Berkeley, CA. 94720

### 1. Introduction

Floating-gate structures which utilize charge tunneling through thin oxides are now widely used for making electrically erasable programmable read-only memories, or EEPROM's[1]. The future trend for this type of non-volatile memory design is likely moving toward high packing density with thinner oxides ( $<100\text{\AA}$ ) as the tunnel barrier. It is important, therefore, to recognize all relevant effects associated with charge tunneling through thin oxide in the MOS system. In particular, we need to understand the tunneling characteristics of thin oxides so that these devices are scaled properly without suffering performance penalty. In this paper, theoretical modeling and experimental results on direct tunneling, tunneling-induced electron-hole pair generation in silicon, and hole tunneling are presented. These studies not only lead us to better understand the fundamental limits in these devices but also provide us with some insights in the physical properties of the Si-SiO<sub>2</sub> materials.

### 2. Direct vs. Fowler-Nordheim Tunneling

Charge retention is a very important consideration in the performance of non-volatile memory. Typically, a minimum of 11 order-of-magnitude difference in current levels is required between programming and read operations. This can readily be achieved in the thicker oxides owing to the exponential current-voltage relationship of Fowler-Nordheim tunneling (Fig.1a). In thinner oxides ( $<60\text{\AA}$ ), however, direct tunneling (Fig.1b) can become important at low voltages[2]. The trapezoid-shaped oxide barrier gives rise to a tunneling current which has relatively weak dependence on oxide field. Using the semi-classical independent electron approach with a Franz-type two-band-like dispersion relation in the WKB approximation of tunneling probability, we have modeled the tunneling current-voltage relationships for a wide range of oxide thicknesses. Some of the theoretical results, together with their experimental data, are illustrated in Fig.2 for electron tunneling from metal gate to a p-type substrate. From Fig.2, it is apparent that the

current changes much slower as a function of gate voltage for oxide voltage below the barrier-height potential (3.2V). This direct tunneling phenomenon has to be considered in scaling the oxides so that no excessive charges can leak away during the read cycles.

### 3. Electron-Hole Pair Generation In Si

Although those electrons participating in tunneling at the cathode are "cold", they do become "hot" when arriving at the anode, with a maximum possible energy equal to the potential drop between the two electrodes. Upon entering the Si, these energetic electrons will lose energy by phonon scattering and by impact ionization[3,4]. In the erase operation of a memory cell, therefore, electron-hole pairs are generated in the  $n^+$  region underneath the tunnel oxide. Some of these generated holes will drift to the Si-SiO<sub>2</sub> interface and flow out into the  $p^+$  channel stop region to become a substrate current. This current can be comparable to the tunneling electron current in magnitude and is generally not desirable in circuit operation.

In this work, Si-gate p-channel MOS transistors are used for studying the carrier multiplication effect (Fig.3). Our experimental set-up is the same as previous investigator's[5,6] in which a negative voltage ramp is applied to the gate electrode and the gate, drain/source and substrate currents are measured. Under most conditions, the gate current ( $I_g$ ) is purely the electron tunneling current and the drain/source current ( $I_{d-s}$ ) is the generated hole current (flowing out of  $p^+$  drain/source), whereas the substrate current ( $I_{sub}$ ) is the combination of the tunneling electron current and the generated electron current. Therefore, the ratio  $I_{d-s}/I_g$  represents the quantum yield ( $\gamma$ ), or number of generated electron-hole pairs per incident electron for a given bias. The quantum yields vs. oxide voltage, which is  $\sim 1.3V$  less than the gate voltage, are plotted in Fig.4 for several different oxides. It is interesting to observe that there exists a threshold ( $\sim 1.7V$ ) for pair generation and the maximum quantum



yield is less than 2, irrespective of the oxide thickness and applied voltage. The threshold effect at  $\sim 1.7$  volts is a direct manifestation of conservation of momentum in silicon[4] in which it requires a minimum of  $3/2$  of the bandgap energy for the incident electron to make impact ionization. The experimental observation that the quantum yields are relatively insensitive to the applied voltages for thicker oxide ( $>100\text{\AA}$ ) samples is indicative of strong lattice scatterings in the insulator. It is believed that hot electrons lose energy and get randomized in  $\text{SiO}_2$  primarily by LO phonon emission[5,6]. Assuming the energy distribution of these hot electrons is a displaced Maxwellian, one can derive an energy conservation equation[7,8] for the transport of the "average" electron in  $\text{SiO}_2$ . This phenomenological equation is expressed in Eq.1.

$$\frac{dE_e}{dx} = qE_{ox} - \frac{E_e}{\lambda} \quad (1)$$

where  $E_e$  is the average electron energy,  $E_{ox}$  is the oxide field and  $\lambda$  is the empirical energy relaxation mean-free-path, respectively. The term on the left-hand side of Eq.1 represents the net energy increase per unit distance. The first term on the right-hand side represents the energy gain from the electric field and the second term represents the energy loss due to collision. The solution to the above first-order linear differential equation is given by Eq.2.

$$E_e(x) = qE_{ox}\lambda \left[ 1 - e^{-\frac{(x-t_{ox})}{\lambda}} \right] + \phi_b, \quad s_{ox} < x < t_{ox} \quad (2)$$

where  $t_{ox}$  is the oxide thickness,  $s_{ox}$  is the tunneling distance in oxide and  $\phi_b$  is the Si-SiO<sub>2</sub> barrier height (3.2eV). The average energy of these electrons in exiting the oxide is simply  $E_e(x=t_{ox})$ . Note that Eq.2 is only good for  $qV_{ox} > \phi_b$  where scatterings of electrons in the conduction band of SiO<sub>2</sub> have occurred. In direct tunneling, on the other hand, no scattering on the tunneling electrons is assumed and the energy gained in exiting the oxide is equal to the oxide potential drop.

Eq.2 says that  $E_e(t_{ox})$  is only a function of the oxide field, provided that  $t_{ox} > \lambda$  (or, the distance that the electrons spent in the conduction band of SiO<sub>2</sub>) is greater than  $\lambda$ . The experimental data on quantum yield are re-plotted in Fig.5 as a function of oxide fields. It can be seen that, in thicker oxide samples, the quantum yields, which are directly related to the incident electron energies, are essentially not thickness dependent, in agreement with our theoretical analysis.

To further extend this work, we have used Eq.2, together with a  $\lambda$  value of 30 Angstroms, to calculate the average energy of electrons incident at the Si surface. Quantum yields are re-plotted in Fig.6 as a function of the calculated average electron energy (relative to the conduction-band

edge of silicon). It compares favorably with the theoretical work on ionization probability reported by Drummond and Moll[4] in the energy range of less than  $\sim 3.5\text{eV}$ . It is interesting in observing an "inflection" in curvature around 4eV. This phenomenon is probably related to the detailed energy-band structure in the silicon. The higher yield exhibited by the 79A oxide sample, of which the thickness is comparable to  $\lambda$ , could be a manifestation of the ballistic effect[8] in SiO<sub>2</sub>. Although this needs to be further investigated.

#### 4. Hole Tunneling

Faraone and Hsueh[9], by using a structure incorporating a p-channel MOSFET with a metal/tunnel-oxide/n-silicon device, have demonstrated that the dominant carrier transport in 20A or thinner SiO<sub>2</sub> (Al-gate) is hole flow under negative bias. In our work on quantum yield, we also observed hole carrier transport through very thin oxides ( $<40\text{\AA}$ ) in the voltage range where carrier multiplication is not significant. In Fig.7, We show the gate and drain-source currents for p-channel MOSFET's of oxide thicknesses 35A and 41A. The p-n diode leakage current of these transistors is  $\sim 10$  femtoamps. For the 35A oxide sample, the drain-source current is initially negative (flowing into the drain/source), indicating hole carriers either transporting through the oxide or recombining with tunneling electrons at the Si surface. For  $V_g > -3\text{V}$  (or,  $V_{ox} > -1.7\text{V}$ ), appreciable holes are generated in the Si channel depletion region by energetic tunneling electrons to cause a decrease in the drain-source current.  $I_{d-s}$  changes sign at  $V_g \sim -3.2\text{V}$  and stays positive beyond this voltage. Notice that the gate current is about one order of magnitude larger than the drain-source current before significant pair generation takes place. This is a good indication that electron current is still the dominant carrier species in the transport of 35A-thick SiO<sub>2</sub> (poly-gate). The 41A oxide data are qualitative the same, except that the hole tunneling current is much smaller.

For Si-gate n-channel MOSFET's, several authors[10,11,12] have observed substrate hole current in thin and thick oxide samples under sufficiently high positive gate bias. They have attributed this current to tunneling by certain kinds of valence-band electrons that leave holes behind, giving rise to a substrate current. We have measured the substrate currents as well as the gate currents for devices with different oxide thicknesses and plot their ratios ( $I_g/I_{sub}$ ) in Fig.8. In the very thin oxide data, a threshold at  $V_g \sim 1.1\text{V}$  for the onset of hole current can be observed. This may correspond to the situation where the valence-band edge at the Si surface is in line with the conduction-band edge of the  $n^+$  poly-gate. Hence, as the gate voltage is increased further, there will be empty states available on the gate side for the Si valence-band electrons to tunnel

into. The ratio ( $I_g/I_{sub}$ ) increases from  $<100$  at low bias to  $>1000$  at medium bias and then decreases at high bias. However, our theoretical simulation based on a 2-band model predicts a ( $I_g/I_{sub}$ ) ratio generally larger than 10,000 for medium and high biases if only the valence-band electron tunneling is considered. This discrepancy points to the possibility of mechanisms other than pure valence-band electron tunneling may be involved. The contributions from tunneling by hot holes generated at the polysilicon surface and from tunneling by field-enhanced excitation of electrons from the Si valence band[12] should be carefully analyzed in order to determine the exact origin of this observed substrate current.

### 5. Summary

We have investigated the carrier tunneling related phenomena, namely, direct tunneling, lattice scattering and field heating in oxide, carrier multiplication in silicon, hole and valence-band electron tunneling in thin-oxide MOSFET's. It is found that direct tunneling imposes a fundamental limit on oxide-thickness scaling. The average energy of the electrons in the oxide conduction band is primarily a function of oxide field as a result of strong lattice scattering. An impact ionization threshold of  $\sim 1.7\text{eV}$  is observed in Si and a quantum yield less than 2 is measured for all the samples. Finally, hole tunneling is observed in both Si-gate p-channel and n-channel devices.

### Acknowledgement

The authors wish to thank Simon Tam for useful discussions and to Simon Law and Ali Ibrahim of Xerox for providing some transistor samples for measurement. Research has been sponsored by DARPA under grant N00039-81-K-0251.

### References

- [1] W.S. Johnson et al., Proc. IEEE ISSCC Conf., p.152, 1980.
- [2] C. Chang et al., Device Research Conf., 1983.
- [3] W. Shockley, Solid-State Electronics, Vol.2, p.35, 1961.
- [4] W.E. Drummond and J.L. Moll, J. Appl. Phys., 42(13), p.5556, 1971.
- [5] T.N. Theis et al., Proc. INFOS Conf., 1983.
- [6] E. Suzuki and Y. Hayashi, J. Appl. Phys., 50(11), p.7001, 1979.
- [7] T. Toyabe and H. Kodera, Japanese J. Appl. Phys., 13(9), p.1404, 1974.
- [8] D.K. Ferry et al., IEEE EDL, 2(9), p.228, 1981.
- [9] L. Faraone et al., Proc. INFOS Conf., 1983.
- [10] B. Eitan and A. Kolodny, Device Research Conf., 1983.
- [11] C. Chang et al., Proc. INFOS Conf., 1983.
- [12] L.D. Yau et al., IEEE EDL, 4(8), p.261, 1983.

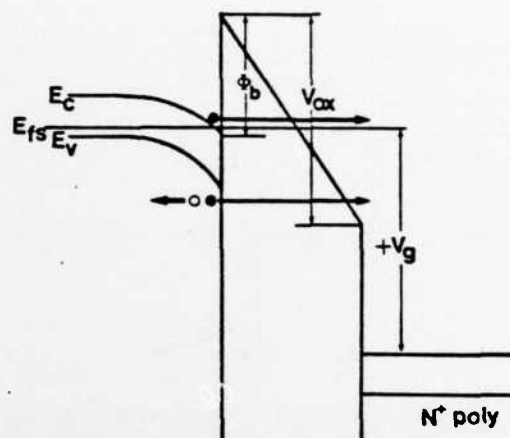


Fig.1a Energy-band diagram showing Fowler-Nordheim tunneling (triangular barrier,  $V_{ox} > \phi_b$ ) from the p-type substrate to the gate of a MOS structure.

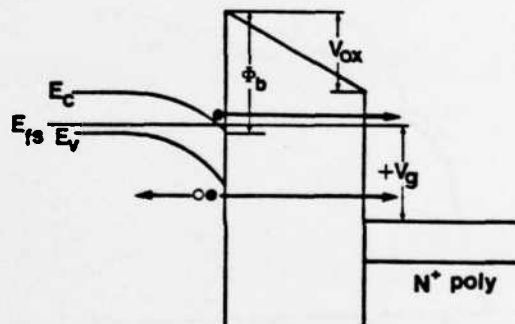


Fig.1b Energy-band diagram showing Direct tunneling (trapezoidal barrier,  $V_{ox} < \phi_b$ ). Electrons do not go into the conduction band of the  $\text{SiO}_2$  in this process.

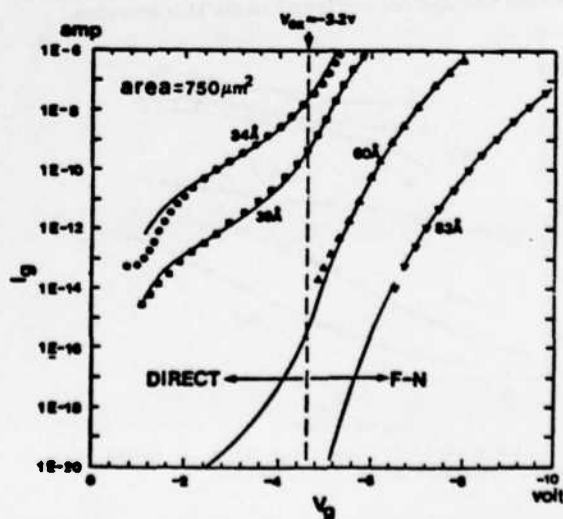


Fig.2 Theoretical and experimental tunneling I-V's of Al-gate n-channel MOS structures under negative gate bias, illustrating the difference in I-V characteristics between F-N and Direct tunneling.

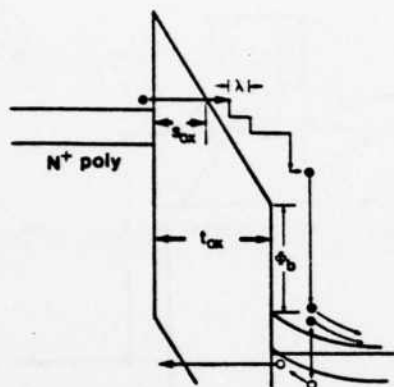


Fig.3 Energy-band diagram of a p-channel MOS device showing tunneling electrons, first get scattered in  $SiO_2$ , then make impact ionizations in Si. Also shown are the holes tunneling in the reverse direction.

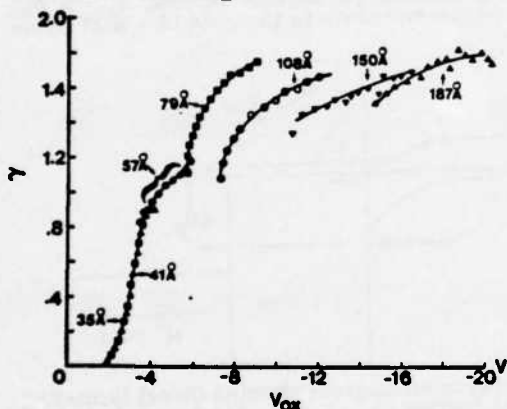


Fig.4 Measured quantum yield ( $\gamma$ ) in silicon as a function of oxide voltage for different oxide thicknesses.

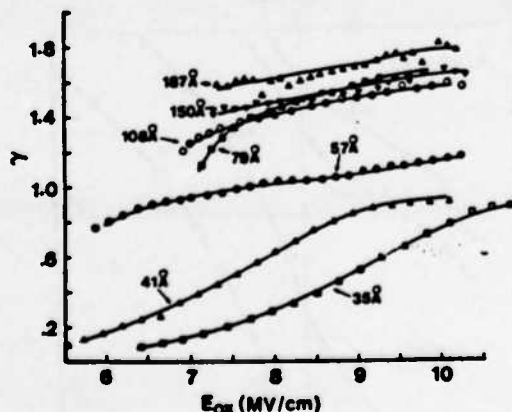


Fig.5 Quantum yield ( $\gamma$ ) as a function of oxide electric field for different oxide thicknesses.

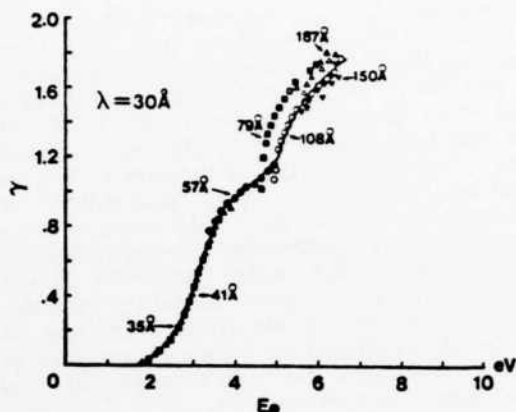


Fig.6 Quantum yield ( $\gamma$ ) vs. average incident electron energy (calculated from Eq.2 assuming a  $\lambda$  of 30 Å). The slope of this curve gives  $\sim 3.7$  eV per pair generation.

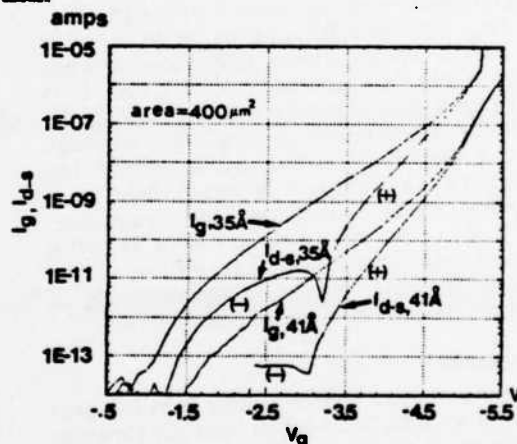


Fig.7 Experimental data of gate current (tunneling current) and drain/source current (hole current) for 35 Å and 41 Å oxides, p-channel MOSFET's under negative gate bias. The (-) hole current indicates direct hole tunneling into gate whereas the (+) hole current indicates the generated hole current flowing out of drain/source.

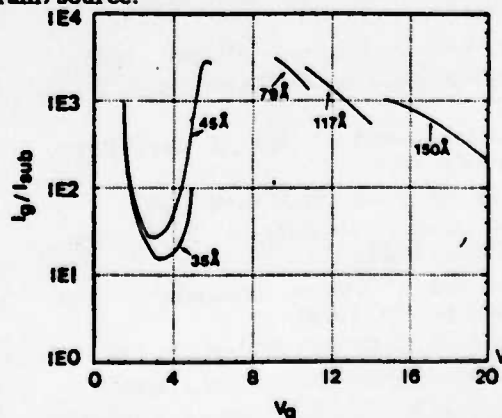


Fig.8 Measured ( $I_g/I_{sub}$ ) ratio vs. gate voltage of n-channel MOSFET's for different oxide thicknesses. The tunneling conditions here correspond to the diagrams shown in Fig.1.



## On Physical Models for Gate Oxide Breakdown

*S. Holland, I.C. Chen, C. Hu and T.P. Ma†*

Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, CA. 94720

### Abstract

Electrical breakdown of thin (32 nm)  $\text{SiO}_2$  films subjected to constant-current stressing is studied. By studying the effects of reversing the polarity of the constant-current bias and the effects of thermal annealing on the charge-to-breakdown it is determined that electrical breakdown of  $\text{SiO}_2$  is not caused by the widely-cited accumulation of trapped electrons. Rather it is caused by the build-up of positive charges near the cathode at localized areas. The positive charges are not mobile ions but exhibit many characteristics of trapped holes. We conclude that electrical breakdown in  $\text{SiO}_2$  is caused by the accumulation of holes, generated by impact ionization in the oxide.

---

† On leave from Yale University.

## 1. Introduction

Time-dependent dielectric breakdown of  $\text{SiO}_2$  may be divided into two stages. The first is a build-up stage during which localized high-field/current-density regions are formed. The brief second stage begins during which runaway electrical and or thermal processes quickly bring the oxide to breakdown. The time required to complete the build-up stage, of course, determines the lifetime of the oxide.

Three basic physical models for the build-up stage are known to us [1-5]. According to one model [1], positive impurity ions, such as  $\text{Na}^+$ , migrate to the cathode interface under the influence of the field, resulting in an increased local field and reduced barrier height at the cathode (Figure 1a). A second model [2-4] is similar to the first except for postulating that the trapped positive charge results from hole generation by impact ionization in the oxide (Figure 1b). A third and widely-cited model [5] suggests that electron trapping causes an increase in the electric field at the anode (for a fixed stressing voltage) (Figure 1c). Breakdown results when the field reaches a critical value at which the Si-O bond is broken. However, conclusive experimental verification for these different models has been lacking.

This paper presents new experimental evidence which supports the hole-build-up model and contradicts the other two models.

## 2. Experimental Results and Discussion

The samples used in this study were polysilicon-gate MOS capacitors. 10-20  $\Omega\text{-cm}$  p-type substrates were used. The gate oxide was grown to a thickness of approximately 32 nm at 900C in dry  $\text{O}_2$ , followed by a 10 minute anneal in nitrogen. Polysilicon was then deposited and implanted with arsenic. The arsenic was driven in at 1000C in dry  $\text{O}_2$  for 1 hour. The post-metallization anneal was

performed at 450C for 15 minutes in forming gas.

The capacitors were biased with a constant current to accelerate breakdown. This technique was first described by Harari [5]. The use of a constant-current stress facilitates the measurement of the "charge-to-breakdown" (the amount of charge passing through the oxide before breakdown). There is no difference in the physics of oxide breakdown caused by constant-voltage stress and by constant-current stress. The current flow in SiO<sub>2</sub> is due to Fowler-Nordheim tunneling, which has an I-V characteristic given by

$$I = A E_c^2 e^{\frac{-B}{E_c}} \quad (1)$$

where A and B depend on the electron effective mass and the barrier height at the injecting interface, i.e., the cathode.  $E_c$  is the cathode electric field.

A small fraction of the electrons injected into the SiO<sub>2</sub> are subsequently trapped. Because the constant-current bias maintains a constant cathode field (Eq. 1), the gate voltage necessary to maintain a constant current increases as the density of trapped electrons increases.

Figure 2 shows typical IV characteristics of a device before and after being subjected to a constant-current stress with the gate biased negative. Figure 2a shows the IV curves for positive  $V_g$  (first quadrant in the IV plane). After stressing, the IV characteristics are shifted to the right, which is indicative of electron trapping in the oxide. Also shown in Figure 2a are the IV characteristics of the stressed device measured after annealing the device in forming gas at 450C. As can be seen, the effect of the anneal is to remove the negative charge from the oxide. The post-anneal IV characteristic is nearly identical to its original value. In Figure 2b, the third quadrant IV characteristics (negative  $V_g$ ,  $I_g$ ) show a dramatic reduction in slope due to stressing. The 450C anneal shifted the IV to the left by removing the trapped electrons as expected. However, the slope

remains reduced, and the current is increased or shifted to the left when compared to the original (before stressing) IV. The direction of the shifts in the IV curves is consistent with positive charge trapping. Furthermore, slope reduction and current increase in the Fowler-Nordheim characteristics can be attributed to cathode barrier lowering due to trapped positive charge near the cathode, i.e. the gate [6]. Therefore we conclude that positive charge accumulates near the gate, i.e. the cathode during and as a result of stressing. The positive charging can be shown to be localized at a small fraction of the area [7], where, perhaps, hole trap density is higher. Hole trapping near the anode (the substrate) is not sufficient to cause significant barrier lowering as the IV slope is little changed after stressing in Figure 2a. The effect of the annealing step was to remove the trapped negative charge, while a large portion of the positive charge remains trapped near the cathode.

The effect of this residual positive charge on the breakdown was then studied. Devices were stressed to approximately 90% of the total charge per unit area,  $Q = J \times t$ , necessary for breakdown for that particular polarity, where  $J$  is the current density, and  $t$  is the stressing time. A current density of  $33 \text{ mA/cm}^2$  was used in this experiment. The devices were then annealed in forming gas. Annealing temperatures of 350C and 450C were used. Then, the additional charge  $Q_{BD}$  necessary for breakdown was measured. The results are shown in Table 1. From rows #3 and 5, one observes that if the same polarity current is employed after annealing, then only a similar small amount of additional charge is necessary for breakdown, whether the devices are annealed at either temperature or not annealed at all after the initial stressing. Also, the total charge-to-breakdown  $Q_{total} = Q_i + Q_{BD}$ , is essentially the same in rows 3 and 2 and in rows 5 and 1.

By these observations, we rule out electron trapping as the cause of the breakdown. If the trapped negative charge was responsible for the breakdown, then one would expect, in row 3 or 5, that the  $Q_{BD}$  after annealing would be much larger than the  $Q_{BD}$  without annealing since annealing had removed the electrons trapped during the initial stress and restored the oxide to the original condition. We conclude that oxide breakdown is due to a build-up of positive charge which is not annealed out at temperatures as high as 450C (see Fig. 2b). In fact, hardly any positive charge is detrapped at 450C as evidenced by the insensitivity of  $Q_{BD}$  to annealing in all the rows. Table 1 also shows that there exists an order of magnitude increase in  $Q_{BD}$  when the final stressing polarity is reversed from the initial stressing polarity (row 4 vs. row 3, and 6 vs. 5). The higher  $Q_{BD}$  due to the reversal of the polarity of the stress (without the thermal anneal) is more clearly illustrated in Figure 3. Without the polarity reversal,  $Q_{total} = Q_i + Q_{BD} = \text{constant}$  as expected. With the polarity reversal,  $Q_{BD}$  is essentially independent of  $Q_i$ . This supports the model that positive charge builds up near the cathode during stress. The final  $Q_{BD}$  is limited by the build-up of positive charge near the final cathode during the final stress and is insensitive to the positive charge trapped near the initial cathode during the initial stress. This results suggests that extrapolation of device lifetime based on charge-to-breakdown tests with a single-polarity current stress is slightly pessimistic.

Another point worth noting is the significant difference in the  $Q_{BD}$  observed when one compares injection from the polysilicon-gate electrode to that of injection from the silicon substrate. Injection from the substrate results in a significantly larger  $Q_{BD}$ , as shown in Figure 3 and Table 1. Similar results have been reported previously [9]. This effect could be explained in terms of a more efficient trapping of the generated positive charge near the polysilicon-  $\text{SiO}_2$

interface when compared to the Si-SiO<sub>2</sub> interface, or in terms of a weaker polysilicon-SiO<sub>2</sub> interface such that the final runaway stage begins at a lower density of trapped positive charge. In fact, the well known localized nature of oxide breakdown (the weak spots) can be understood in these terms, too.

In order to rule out mobile ion contamination as the cause of the breakdown, several devices were stressed to  $14\text{ C/cm}^2$  in one polarity, an additional  $14\text{ C/cm}^2$  in the opposite polarity, and then stressed to breakdown in the original polarity. The final  $Q_{BD}$  was always of the order of  $1\text{ C/cm}^2$ . The fact that the device breaks down quickly in the third stress suggests that the positive charge that is trapped at the original cathode basically remains trapped there. If mobile ions were the positive charge responsible for the breakdown then one would expect a larger  $Q_{BD}$  at the third step because the ions accumulated during the first stress would have been moved away (from the cathode) by the second stress. Certainly the presence of mobile ions such as Na<sup>+</sup> in large amounts in the oxide will accelerate the breakdown process, as has been shown several times [1,9-11]. What this experiment proves is that the minute concentrations of Na<sup>+</sup> present in today's MOS processes is not responsible for oxide breakdown.

All of the data presented here points towards the accumulation of positive charge in the SiO<sub>2</sub> near the cathode during the high-field constant-current stress. The positive charge is not mobile ions. We believe that the positive charge is holes generated by impact ionization in the oxide and is then drifted to the cathode, where some of the holes are trapped. The reasons are presented below.

Recently Nissan-Cohen et al reported the generation of positive charge in SiO<sub>2</sub> films [12]. They concluded that the most likely origin for this positive charge was impact ionization. In our study, we found electrical and annealing

behavior of the trapped positive charge to be similar to that of radiation-induced positive charge [13]. As is well known, ionizing radiation results in the formation of hole-electron pairs in the  $\text{SiO}_2$ , with the created holes being trapped at the interfaces with a fairly high probability. Thus this similarity in behavior of positive charge generated by ionizing radiation and high-field stressing offers further support for impact ionization in the  $\text{SiO}_2$ . For example, Figure 4 shows some typical results of the annealing of the damage caused by the constant-current injection. The quasi-static CV measurements of a device stressed to  $+18 \text{ C/cm}^2$  and then successively annealed at temperatures of 350C and 450C are compared with the characteristics of the device before stress. After the first anneal, the CV characteristics are shifted to the left indicating the presence of residual positive charge. In addition, fast surface states are present. After annealing at 450C, the positive charge density is reduced and the surface states are removed. This type of behavior is consistent with reports of the annealing of radiation-induced trapped holes at the  $\text{SiO}_2$  interface [13-15].

### 3. Conclusion

Experimental evidence has been presented to rule out two previous models of breakdown in  $\text{SiO}_2$ , the electron-trapping model and the mobile-ion model. The breakdown is due to a build-up of positive charge at the cathode interface in localized areas. This positive charge is probably holes generated by impact ionization in the oxide that drift to the cathode to be trapped. The trapped positive charge increases the local field and current density (through barrier lowering) and further accelerates the build-up until a very brief run-away process brings the oxide to destructive breakdown.



#### 4. Acknowledgements

Helpful discussions with S. Tam, M.S. Liang, C. Chang, and Prof. R.S. Muller of UC-Berkeley are gratefully acknowledged. This research was supported by DARPA contract N00039-K-0251 and a grant from Rockwell International Corporation under the MICRO project of the state of California.

## References

- [1] T.H. DiStefano, "Dielectric breakdown induced by sodium in MOS structures", J. Appl. Phys., 44 (1), p. 527, Jan. 1973.
- [2] J.J. O'Dwyer, "Theory of high field conduction in a dielectric", J. Appl. Phys., 40 (10), p. 3887, Sept. 1978.
- [3] T.H. DiStefano and M. Shatzkes, "Impact ionization model for dielectric instability and breakdown", Appl. Phys. Lettr., 25 (12), p. 685, Dec. 1974.
- [4] N. Klein and P. Solomon, "Current runaway in insulators affected by impact ionization and recombination", J. Appl. Phys., 47 (10), p. 4364, Oct. 1976.
- [5] E. Harari, "Dielectric breakdown in electrically stressed thin films of thermal SiO<sub>2</sub>", J. Appl. Phys., 49 (4), p. 2478, April 1978.
- [6] C.M. Osburn and N.J. Chou, "Accelerated dielectric breakdown of silicon dioxide films", J. Electrochem. Soc., 120 (10), p. 1377, Oct. 1973.
- [7] To be published.
- [8] J.J. van der Schoot and D.R. Wolters, "Current induced dielectric breakdown", Conf. Proc. on Insulating Films On Silicon, p. 270, 1983.
- [9] S.I. Raider, "Time-dependent breakdown of silicon dioxide films", Appl. Phys. Lettr., 23 (1), p. 34, July 1973.
- [10] C.M. Osburn and D.W. Ormond, "Sodium-induced barrier-height lowering and dielectric breakdown on SiO<sub>2</sub> films on silicon", J. Electrochem. Soc., 121 (9), p. 1195, Sept. 1974.
- [11] S.P. Li et al, "Time-dependent MOS breakdown", Solid-State Electronics, 19, p. 235, 1976.

- [12] Y. Nissan-Cohen et al, "High field current induced-positive charge transients in  $\text{SiO}_2$ ", J. Appl. Phys., 54 (10), p. 5793, Oct. 1983.
- [13] J.M. Aitken, "Annealing of radiation-induced positive charge in MOS devices with aluminum and polysilicon gate contacts", J. of Electronic Materials, 9 (3), p. 639, 1980.
- [14] A. Reisman and C.J. Merz, "The effects of pressure, temperature, and time on the annealing of ionizing radiation induced insulator damage in n-channel IGFET's", J. Electrochem. Soc., 130 (6), p. 1384, June 1983.
- [15] M. Shimaya et al, "Electron beam induced damage in poly-si gate MOS structures and its effect on long-term stability", J. Electrochem. Soc., 130 (4), p. 945, April 1983.

		Final $Q_{BD}$ $C/cm^2$		
	Initial $Q_i C/cm^2$	No Anneal	350C Anneal	450C Anneal
1	0	+23.9	+22.3	+22.9
2	0	-19.1	-16.3	-18.8
3	-17.0	-1.4	-0.8	-1.2
4	-17.0	+15.4	+13.9	+16.2
5	+20.0	...	+1.4	+1.5
6	+20.0	-11.8	-11.1	-13.5

Table I.

## 5. Figure captions

Fig. 1)

The energy band diagram in the  $\text{SiO}_2$  when:

- a)  $\text{Na}^+$  ions are trapped near the cathode.
- b) holes are trapped near the cathode.
- c) a sheet charge of electrons are trapped in the middle of the oxide.

Fig. 2)

IV characteristics of a device stressed to  $-18.2 \text{ C/cm}^2$  (gate injection, i.e., gate negative).

- a) Substrate-injection (gate positive) IV characteristic.
- b) Gate-injection (gate negative) IV characteristic.

Table I.

Additional charge per unit area necessary for breakdown  $Q_{BD}$  as a function of the initial charge per unit area passed through the oxide  $Q_i$  and the annealing temperature. - refers to electron injection from the polysilicon gate, and + refers to injection from the silicon substrate. ( $J = 33 \text{ mA/cm}^2$ ).

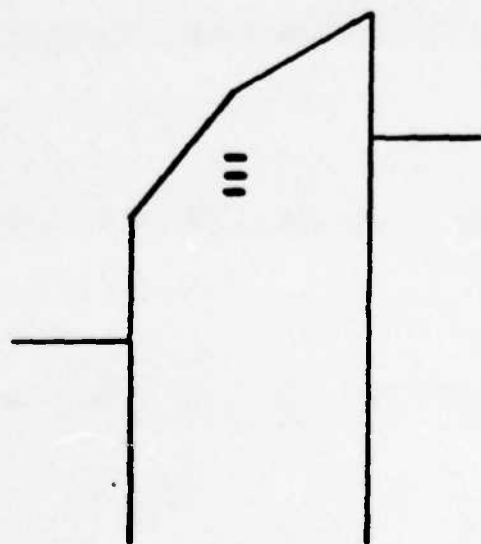
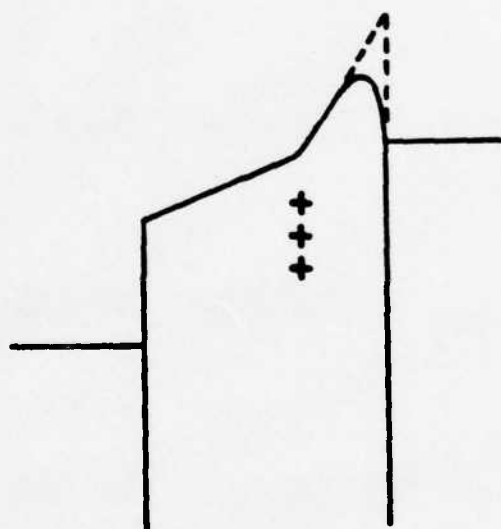
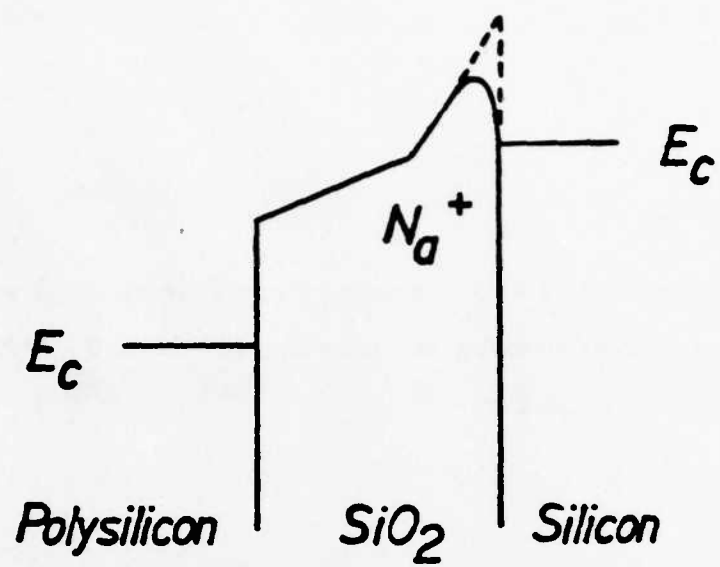
Fig. 3)

Additional charge per unit area  $Q_{BD}$  necessary for breakdown versus the initial charge per unit area  $Q_i$  passed through the oxide. Negative refers to electron injection from the polysilicon gate, and positive refers to injection from the silicon substrate. The current density was  $33 \text{ mA/cm}^2$ .

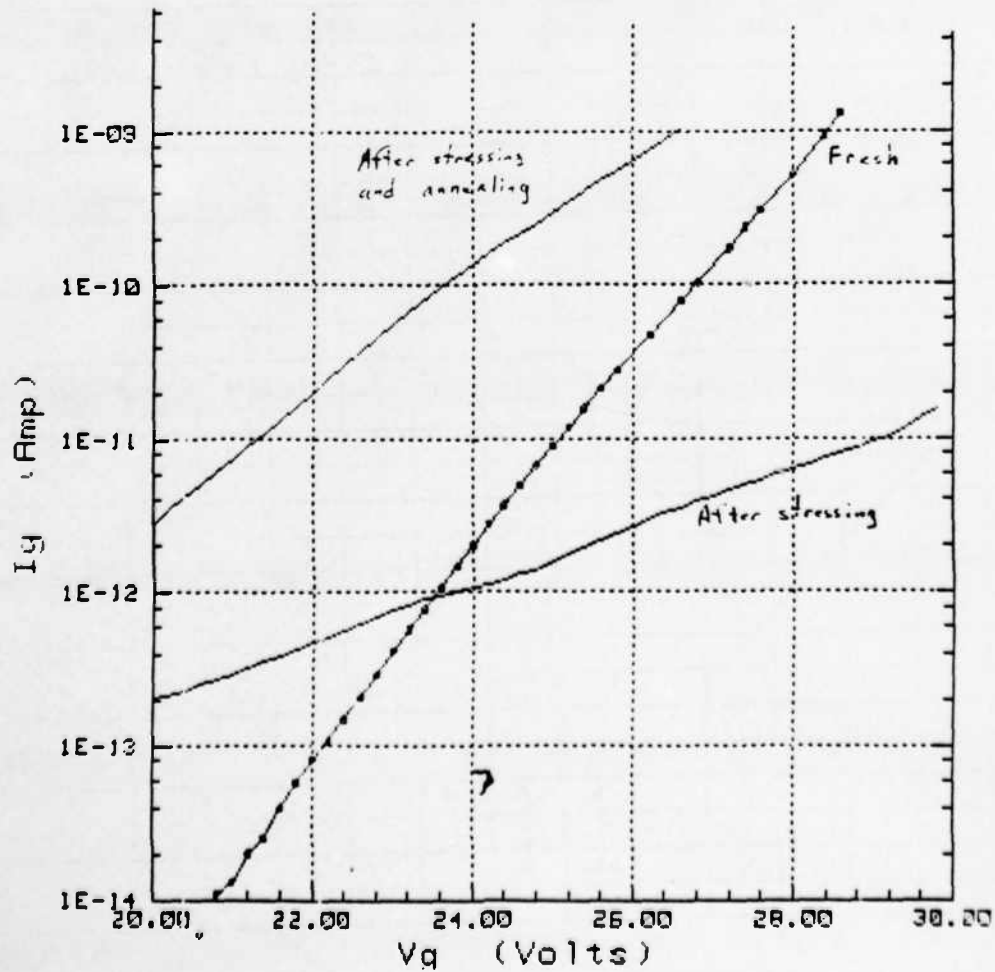
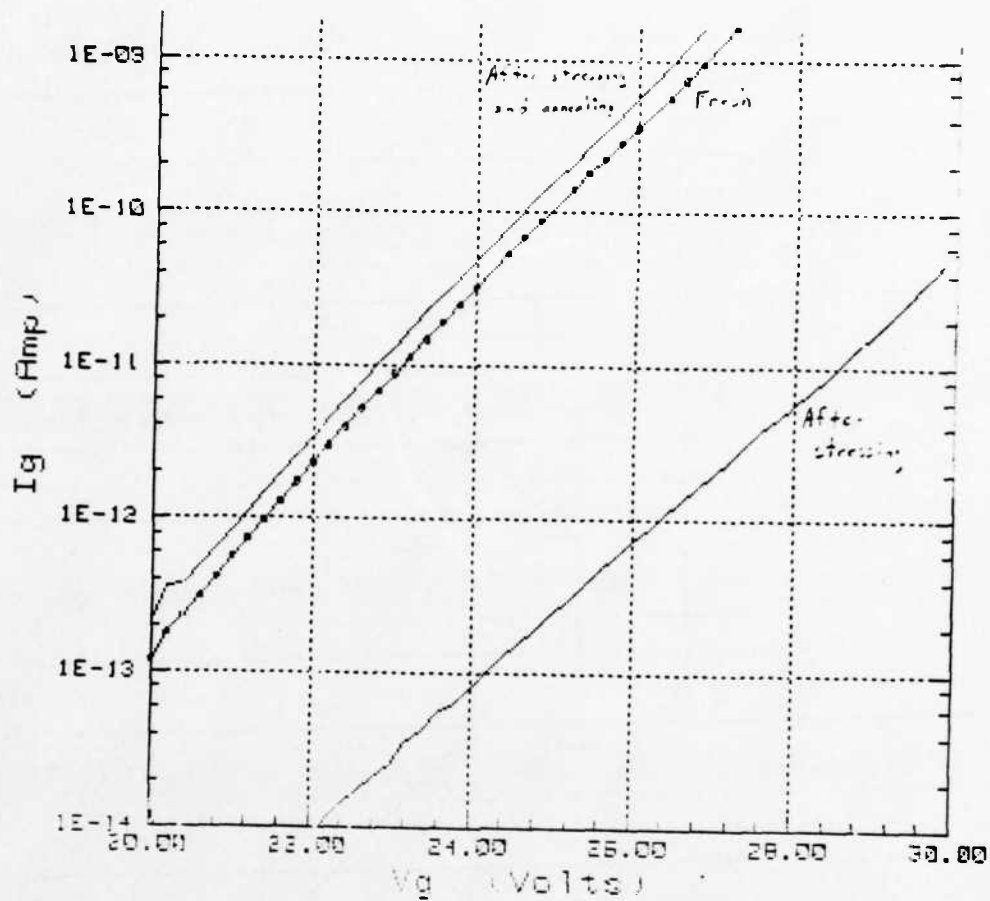
Fig. 4)

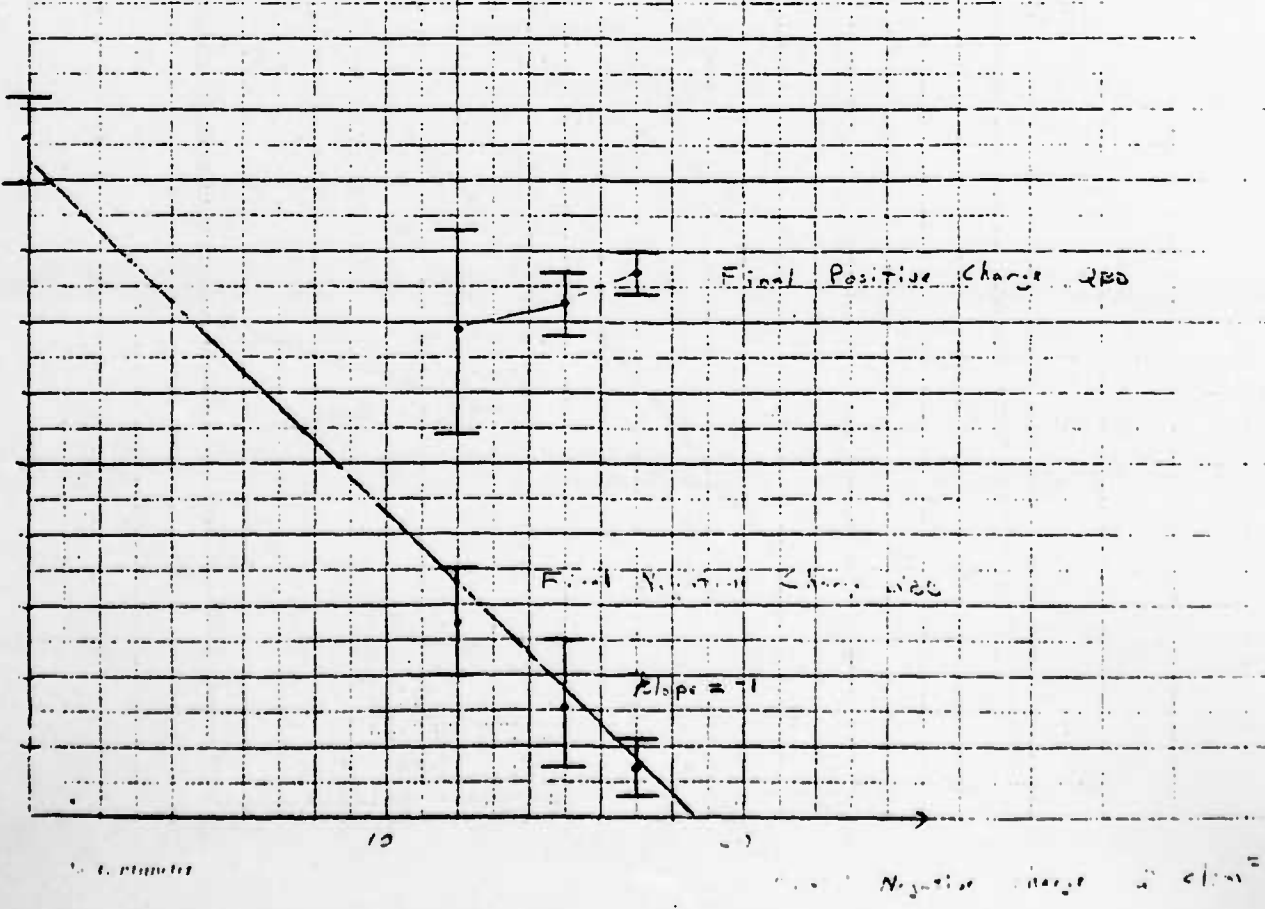
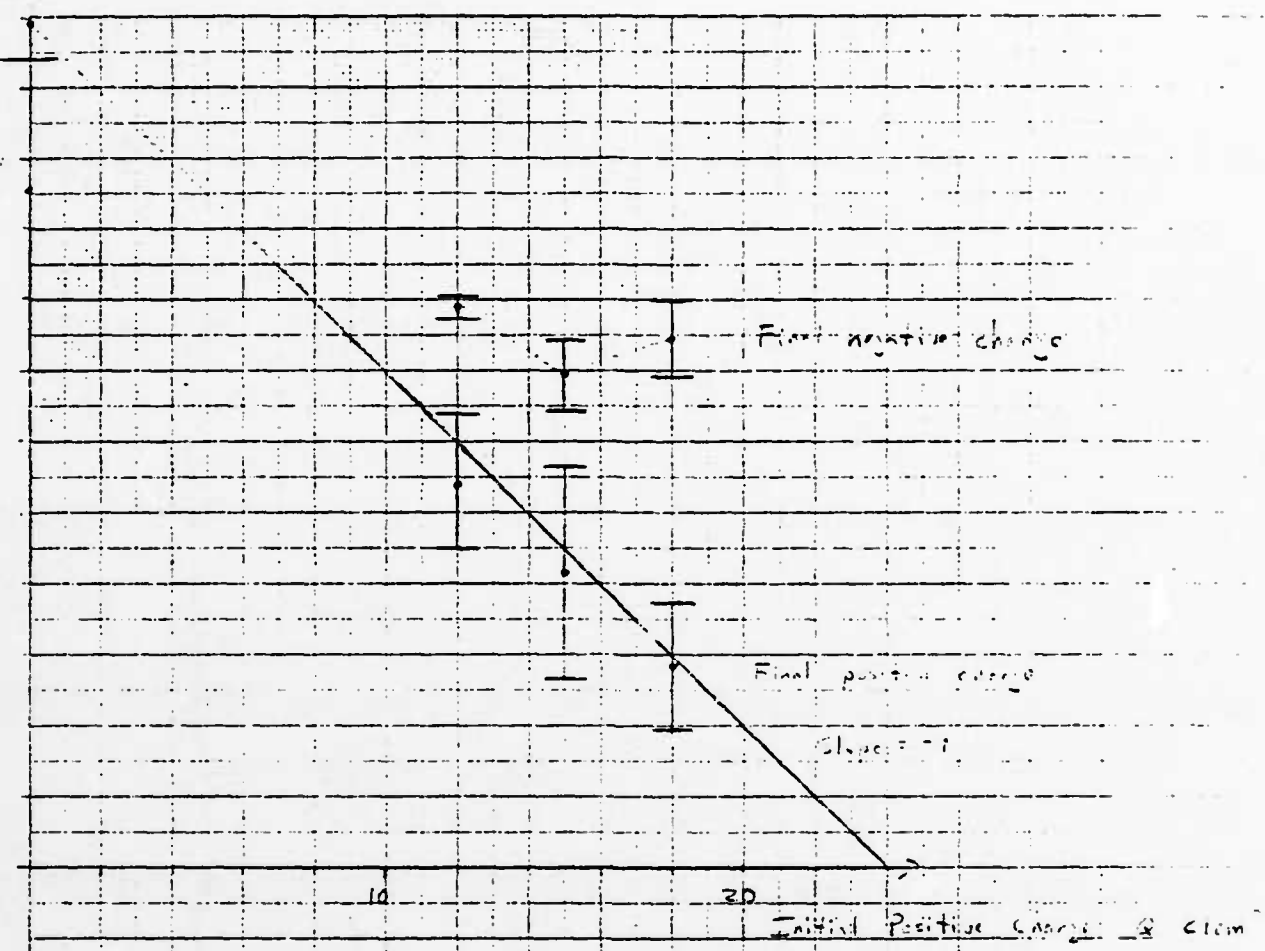
Quasi-static CV characteristics before constant-current stressing (1) and after both constant-current stressing and annealing (2 - 350C, 3 - 450C).

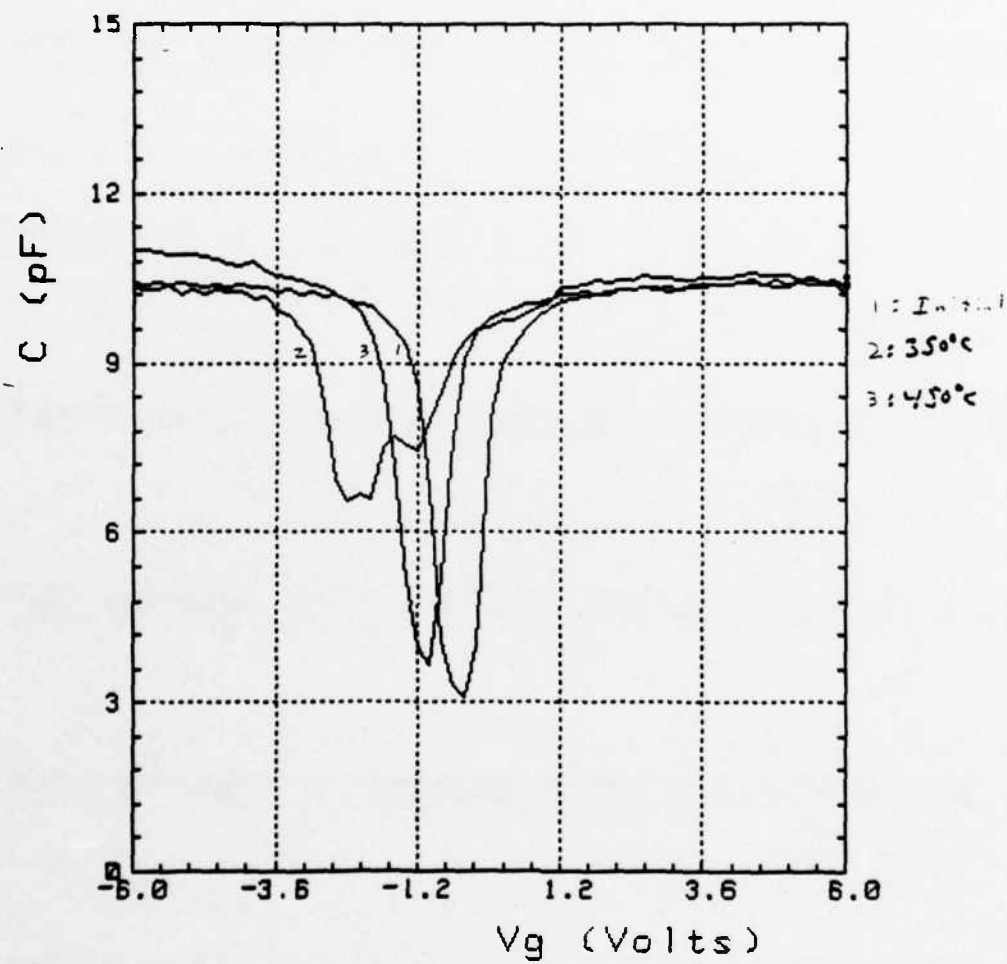
$Q = +18 \text{ C/cm}^2$  (substrate injection).











# Circuits and Systems Letters

## Modeling the Switch-Induced Error Voltage on a Switched-Capacitor

BING J. SHEU AND CHEN MING HU

**Abstract**—An analytical model for switch-induced error voltage on a switched capacitor is derived. A compact expression contains the effects of gate voltage falling rate, threshold voltage, and storage capacitance. It can be used to quickly predict the error voltage. The model is in good agreement with computer simulations using SPICE program and experiment.

### LIST OF SYMBOLS

$\beta$	Conductance coefficient.
$C_L$	Storage capacitance.
$C_{ol}$	Gate-drain overlap capacitance.
$C_{ox}$	Gate capacitance (excluding overlap capacitance).
$C'_{ox}$	Gate capacitance per unit area.

$G$	Channel conductance.
$L$	Effective channel length.
$L_D$	Lateral diffusion distance.
$N_{SUB}$	Substrate doping.
$t_{ox}$	Gate oxide thickness.
$U$	Gate voltage falling rate.
$V_G$	Gate voltage.
$V_H$	High value of $V_G$ .
$V_L$	Low value of $V_G$ .
$V_S$	Signal voltage at the source.
$V_{T0}$	Zero-bias threshold voltage.
$V_T$	Threshold voltage with back-gate bias.
$v_d(t)$	error voltage at drain end at time $t$ .
$v_{dn}$	error voltage at drain end after gate voltage reaches $V_L$ .
$v_{dm}$	Absolute value of $v_{dn}$ .
$W$	Channel width.

### I. INTRODUCTION

In the design of precision MOS analog circuits, particularly switched-capacitor circuits, it is necessary to take into account the switch-induced error voltage on a switched capacitor [1]. An MOS transistor holds mobile charges in its channel when it is on. When the transistor turns off, some portion of the mobile charges

Manuscript received June 21, 1983; revised August 10, 1983. This work was supported by Micro project under Semiconductor Research Corporation and DAPRA under Contract N00039-81-K-0251.

The authors are with the Department of Electrical Engineering and Computer Sciences and Electronics Research Laboratory, University of California, Berkeley, CA 94720.

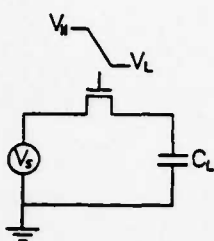


Fig. 1. Schematic of the switch circuit under study.

is transferred to the storage capacitor and cause an error to the sampled voltage (see Fig. 1). The clock voltage feedthrough through the gate-drain overlap capacitance also contributes to the error. The turnoff of an MOS switch consists of two distinct phases. During the first phase, the transistor is on and a conduction channel extends from the source to the drain of the transistor. As the gate voltage falls, mobile charges exit through both the source end and the drain end. In the presence of error voltage there is also a conduction current flowing through the channel between the source and the drain ends. When the gate voltage reaches the threshold voltage, the conduction channel disappears (subthreshold conduction is not included in our model), and the transistor enters the second phase of turnoff. During this phase, only the clock feedthrough through the gate-drain overlap capacitance continues to raise the error voltage. Compensation schemes [2], [3] have been used to reduce the error voltage. However, no analytical expression is available in the literature for quick prediction of the error voltage and the guidance of circuit design.

In this paper, an analytical expression for the switch-induced error voltage is derived. Computer simulations and experiment are used to support the result.

## II. ANALYTICAL MODEL OF ERROR VOLTAGE

We assume that charge pumping phenomenon [4] due to the capture of channel charges by the interface traps is not significant. In other words, when the transistor turns off, all the channel mobile charges exit through the source and drain ends. The circuit schematic to be analyzed is shown in Fig. 1 (nMOS switch is used for illustration). The source end of the switch is connected to a signal voltage source with value  $V_S$ , and the drain end is connected to a storage capacitor with capacitance  $C_L$ . The equivalent lumped models for the circuit during the first and second phases of turnoff are shown in Fig. 2(a) and 2(b), respectively. This lumped model was derived from an exact analysis of the distributed MOSFET model [5].

From the KCL law

$$C_L \frac{dv_d}{dt} = -i'_d + \left(C_{ol} + \frac{C_{ox}}{2}\right) \frac{d(V_G - v_d)}{dt}. \quad (1)$$

We assume that the gate voltage is a ramp function which begins to fall at time 0 from the high value  $V_H$  toward the low value  $V_L$  at a falling rate  $U$ .

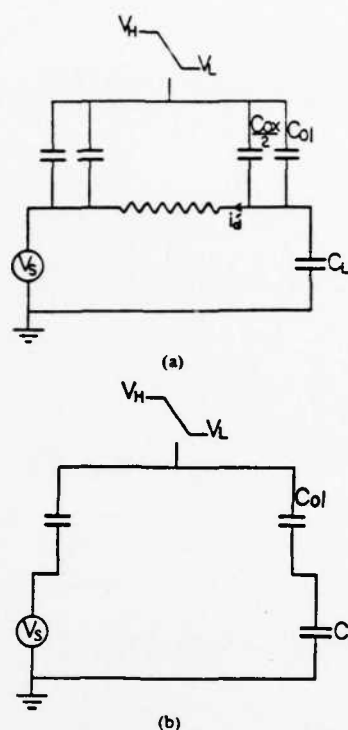
$$V_G = V_H - Ut. \quad (2)$$

Under the condition  $|dV_G/dt| \gg |dv_d/dt|$ , (1) simplifies to

$$C_L \frac{dv_d}{dt} = -i'_d - \left(C_{ol} + \frac{C_{ox}}{2}\right) U. \quad (3)$$

When the transistor is operated in the strong inversion region ( $V_H > V_G > V_S + V_T$ ),

$$i'_d = Gv_d = \beta(V_{HT} - Ut)v_d \quad (4)$$

Fig. 2. Equivalent lumped model for the circuit shown in Fig. 1. (a)  $V_H > V_G > V_S + V_T$ . (b)  $V_S + V_T > V_G > V_L$ .

where

$$\beta = \mu C_{ox} \frac{W}{L} \quad \text{and} \quad V_{HT} = V_H - V_S - V_T$$

(3) becomes

$$C_L \frac{dv_d}{dt} = -\beta(V_{HT} - Ut)v_d - \left(C_{ol} + \frac{C_{ox}}{2}\right)U. \quad (5)$$

The solution of the differential equation is

$$v_d(t) = -\sqrt{\frac{\pi U C_L}{2\beta}} \left( \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right) \exp \left\{ \frac{\beta U}{2 C_L} \left( t - \frac{V_{HT}}{U} \right)^2 \right\} \cdot \left\{ \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right] - \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} (V_{HT} - Ut) \right] \right\}. \quad (6)$$

At  $t' = V_{HT}/U$ , the threshold condition is reached ( $V_G = V_S + V_T$ ) and the first phase ends. After that only the gate-drain overlap capacitor continues to contribute to the error voltage. The error voltage at this time is

$$v_d(t') = -\sqrt{\frac{\pi U C_L}{2\beta}} \left( \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right) \operatorname{erf} \left( \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right). \quad (7)$$

When the gate voltage reaches its final value  $V_L$ , the total amount of switch-induced error voltage on a switched capacitor is

$$v_{dn} = -v_{dm} = -\sqrt{\frac{\pi U C_L}{2\beta}} \left( \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right) \operatorname{erf} \left( \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right) - \frac{C_{ol}}{C_L} (V_S + V_T - V_L). \quad (8)$$

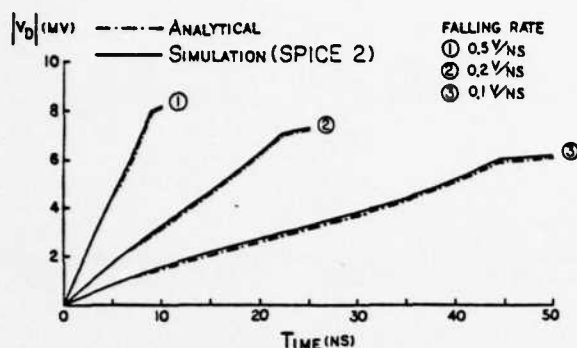


Fig. 3. Comparison of the analytic and computer simulated transient responses for three different gate voltage falling rates. The parameters for Figs. 3 and 4 are  $V_S = 0$ ,  $C_L = 2$  pF,  $V_{TO} = 0.60$  V,  $W = 4$   $\mu$ m,  $L = 3.3$   $\mu$ m,  $L_D = 0.35$   $\mu$ m, and  $\beta = 30.3$   $\mu$ A/V<sup>2</sup>.

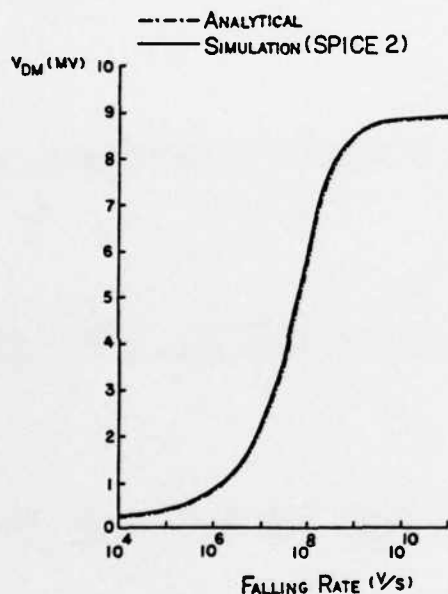


Fig. 4. Comparison of the analytic and computer simulated results of the error voltage as a function of the gate voltage falling rate.

### III. COMPARISON WITH COMPUTER SIMULATION

To validate the model, computer simulations using the SPICE 2G [6], [7] circuit-simulation program have been performed. The circuit configuration for computer simulations is the same as that of Fig. 1.

The analytical transient response, (6), and computer simulated results for three different gate voltage falling rates 0.1 V/ns, 0.2 V/ns, and 0.5 V/ns are shown in Fig. 3. The close agreement between the analytical analysis and the computer simulation is evident. Another comparison is shown in Fig. 4. The error voltage is plotted against the gate voltage falling rate for both the analytical and simulation results. Fig. 5 shows the measured data and calculated result from the analytical model (8). Good agreement is found.

### IV. CONCLUSION

An analytical expression for the switch-induced error voltage on a switched capacitor is presented. Computer simulation and experiment justify the validity of the analysis. The compact

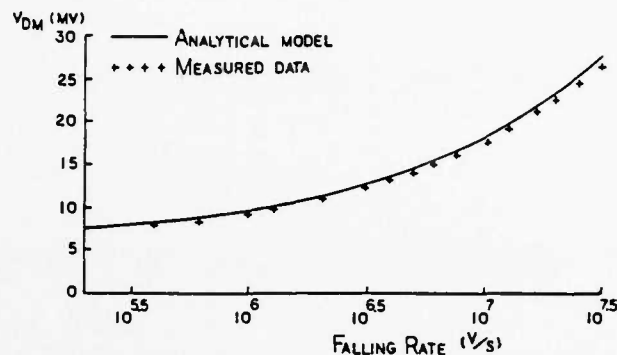


Fig. 5. Measured and calculated error voltages as functions of gate voltage falling rate. The parameters are  $V_S = 0$ ,  $C_L = 24.5$  pF, effective  $C_{ot} = 195$  fF (including parasitic probe capacitance),  $V_{TO} = 0.70$  V,  $W = 40$   $\mu$ m,  $L = 5.1$   $\mu$ m,  $L_D = 0.45$   $\mu$ m,  $t_{ox} = 85$  nm, and  $\beta = 295$   $\mu$ A/V<sup>2</sup>.

expression (8) should be convenient in the analysis of switched-capacitor circuits, such as A/D, D/A converters, and filters.

### ACKNOWLEDGMENT

The authors wish to thank Paul R. Gray and Don O. Pederson for their continuous encouragement and advice. Many helpful discussions from Cheng C. Shih and Ping W. Li are appreciated.

### REFERENCES

- [1] P. R. Gray, D. A. Hodges, and R. W. Brodersen, (Eds.), *Analog MOS Integrated Circuits*. New York: IEEE Press, 1980.
- [2] R. E. Suarez, P. R. Gray, and D. A. Hodges, "All-MOS charge redistribution analog-to-digital conversion techniques: Part II," *IEEE J. Solid-State Circuits*, vol. SC-10, pp. 379-385, Dec. 1975.
- [3] R. C. Yen and P. R. Gray, "A MOS switched-capacitor instrumentation amplifier," *IEEE J. Solid-State Circuits*, vol. SC-17, pp. 1008-1013, Dec. 1982.
- [4] D. Schmitt and G. Dorda, "Interface states in MOSFETs due to hot-electron injection determined by the charge pumping technique," *Electron. Lett.*, vol. 17, no. 20, Oct. 1981.
- [5] B. J. Sheu and C. Hu, "Analysis of the switched-induced error voltage on a switched capacitor," to be published.
- [6] L. W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Electron. Res. Lab., Univ. California, Berkeley, Memo. ERL-M520, May 1975.
- [7] A. Vladimirescu and S. Liu, "The simulation of MOS integrated circuits using SPICE2," Electron. Res. Lab., Univ. California, Berkeley, Memo. ERL-M80/7, Oct. 1980.



## Switch-Induced Error Voltage on a Switched Capacitor

*Bing J. Sheu, and Chenming Hu*

Department of Electrical Engineering and Computer Sciences  
and Electronics Research Laboratory  
University of California  
Berkeley, CA 94720

### ABSTRACT

A concise analytical expression for switch-induced error voltage on a switched capacitor is derived from the distributed MOS-FET model. The result, however, can be interpreted in terms of a simple lumped equivalent circuit. With this expression we explore the dependence of the error voltage on process, switch turn-off rate, source resistance, and other circuit parameters. These results can be used to quickly predict the error voltage. The analytical expression is supported by the close agreement with computer simulations and experiments.

January 11, 1984

---

Research sponsored by Micro project under Semiconductor Res. Corp. and DARPA under contract N00039-81-K-0251.



## Switch-Induced Error Voltage on a Switched Capacitor

*Bing J. Sheu, and Chenming Hu*

Department of Electrical Engineering and Computer Sciences  
and Electronics Research Laboratory  
University of California  
Berkeley, CA 94720

### LIST OF SYMBOLS

$\beta$	Conductance coefficient.
$C_L$	Storage capacitance.
$C_{ol}$	Gate-drain overlap capacitance.
$C_{ox}$	Gate capacitance (excluding overlap capacitance).
$C'_{ox}$	Gate capacitance per unit area.
$G$	Channel conductance.
$L$	Effective channel length ( $L_{DRAWN} - 2L_D$ ).
$L_D$	Lateral diffusion distance.
$N_{SUB}$	Substrate doping.
$R_S$	Source resistance of the signal voltage source.
$t_{ox}$	Gate oxide thickness.
$U$	Gate voltage falling rate.
$V_G$	Gate voltage.
$V_H$	High value of $V_G$ .
$V_L$	Low value of $V_G$ .
$V_S$	Signal voltage at the source.
$V_T$	Threshold voltage with back-gate bias.
$V_{T0}$	Zero-bias threshold voltage.
$v_d(t)$	Error voltage at drain end at time $t$ .
$v_{dn}$	Error voltage at drain end after gate voltage reaches $V_L$ .
$v_{dn}$	Absolute value of $v_{dn}$ .
$W$	Channel width.

## 1. Introduction

The error voltage induced by the turning-off of an MOS switch is one of the fundamental factors that limit the accuracy of switched-capacitor circuits, such as A/D, D/A converters, and filters [1]. An MOS transistor holds mobile charges in its channel when it is on. When the transistor turns off, some portion of the mobile charges is transferred to the storage capacitor and causes an error to the sampled voltage (see Fig. 1). The clock voltage feedthrough through the gate-drain overlap capacitance also contributes to the error. The turn-off of an MOS switch consists of two distinct phases. During the first phase, the transistor is on and a conduction channel extends from the source to the drain of the transistor. As the gate voltage falls, mobile charges exit through both the source end and the drain end. When the gate voltage reaches the threshold voltage, the conduction channel disappears (subthreshold conduction is not included in our analytical analysis), and the transistor enters the second phase of turn-off. During this phase, only the clock feedthrough through the gate-drain overlap capacitance continues to increase the error voltage. The switch-induced error voltage on a switched capacitor can be reduced by turning off the switch very slowly to allow charges return to the source end and by minimizing the part of gate voltage swing that is below the threshold voltage to minimize the effect of the gate-drain overlap capacitance. Compensation schemes [2], [3] have been used to reduce the switch-induced error voltage. However, little work has been done on the analysis of this phenomenon.

In this paper, we analyze the switching-off behavior of the MOS switch. An analytical expression for the switch-induced error voltage is derived. Using this expression we explore the dependence of the error voltage on process and gate voltage falling rate. These results can be used to quickly predict the error voltage. Finally, computer simulations and experiments are used to validate the

analysis. The derivation of the lumped model from the distributed model is attached as appendix A.

## II. Analytical Model of Error Voltage

We assume that the charge pumping phenomenon [4] due to the capture of channel charges by the interface traps is not significant. In other words, when the transistor turns off, all the channel mobile charges exit through the source and drain ends. The circuit schematic to be analyzed is shown in Fig. 1 (NMOS switch is used for illustration). The source end of the switch is connected to a signal voltage source with value  $V_S$ , and the drain end is connected to a storage capacitor with capacitance  $C_L$ . The equivalent lumped models for the circuit during the first and second phases of turn-off are shown in Fig. 2. The configuration of the lumped model is not arbitrarily chosen but results from an exact analysis of the distributed MOSFET model as shown in the appendix.

From the KCL law

$$C_L \frac{dv_d}{dt} = -i_d + (C_{ol} + \frac{C_{ox}}{2}) \frac{d(V_G - v_d)}{dt} \quad (1)$$

We assume that the gate voltage is a ramp function which begins to fall at time 0 from the high value  $V_H$  toward the low value  $V_L$  at a falling rate  $U$ .

$$V_G = V_H - U t \quad (2)$$

Under the condition  $\left| \frac{dV_G}{dt} \right| \gg \left| \frac{dv_d}{dt} \right|^{**}$ , (1) simplifies to

$$C_L \frac{dv_d}{dt} = -i_d - (C_{ol} + \frac{C_{ox}}{2}) U \quad (3)$$

<sup>\*\*</sup> This assumption is also needed in deriving the lumped model. See appendix.

When the transistor is operated in the strong inversion region ( $V_H \geq V_G \geq V_S + V_T$ ),

$$i_d = G v_d = \beta (V_{HT} - U t) v_d \quad (4)$$

where  $\beta = \mu C_{ox} \frac{W}{L}$  and  $V_{HT} = V_H - V_S - V_T$

(3) becomes

$$C_L \frac{dv_d}{dt} = -\beta (V_{HT} - U t) v_d - (C_{ol} + \frac{C_{ox}}{2}) U \quad (5)$$

The solution of the differential equation is

$$v_d(t) = -\sqrt{\frac{\pi U C_L}{2\beta}} \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \exp \left\{ \frac{\beta U}{2 C_L} \left[ t - \frac{V_{HT}}{U} \right]^2 \right\} \cdot \left\{ \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right] - \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} (V_{HT} - U t) \right] \right\} \quad (6)$$

At  $t' = \frac{V_{HT}}{U}$ , the threshold condition is reached ( $V_G = V_S + V_T$ ) and the first phase ends. After that only the gate-drain overlap capacitor continues to contribute to the error voltage. The error voltage at this time is

$$v_d(t') = -\sqrt{\frac{\pi U C_L}{2\beta}} \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right] \quad (7)$$

When the gate voltage reaches its final value  $V_L$ , the total amount of switch-induced error voltage on a switched capacitor is

$$v_{dn} \equiv -v_{dm} = -\sqrt{\frac{\pi U C_L}{2\beta}} \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \operatorname{erf} \left[ \sqrt{\frac{\beta}{2 U C_L}} V_{HT} \right] - \frac{C_{ol}}{C_L} (V_S + V_T - V_L) \quad (8)$$

- 5 -

It is well known that

$$\text{erf}(x) \approx \begin{cases} 1 & \text{if } x \gg 1 \\ \frac{2x}{\sqrt{\pi}}(1 - \frac{x^2}{3}) & \text{if } x \ll 1 \end{cases}$$

therefore, expression (8) can be simplified under the two extreme cases.

for slow switching-off,  $\frac{\beta V_{HT}^2}{2C_L} \gg U$

$$v_{dm} = \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \sqrt{\frac{\pi U C_L}{2\beta}} + \frac{C_{ol}}{C_L} (V_S + V_T - V_L) \quad (9)$$

for fast switching-off,  $\frac{\beta V_{HT}^2}{2C_L} \ll U$

$$v_{dm} = \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \left( V_{HT} - \frac{\beta V_{HT}^2}{6U C_L} \right) + \frac{C_{ol}}{C_L} (V_S + V_T - V_L) \quad (10)$$

### III. Dependence on Process and Electrical Parameters

The switch-induced error voltage on a switched capacitor is affected by many factors. The factors of the greatest interest are the gate voltage falling rate, signal voltage level, substrate doping, oxide thickness, transistor size, and source resistance of the signal voltage source. Common parameter values used in the following examples are:  $W = 4 \mu\text{m}$ ,  $L = 3.3 \mu\text{m}$ ,  $L_D = 0.35 \mu\text{m}$ ,  $C_L = 2 \text{pf}$ ,  $t_{ox} = 70 \text{nm}$ ,  $N_{SUB} = 5.0 \cdot 10^{14} \text{cm}^{-3}$ ,  $V_{T0} = 0.8 \text{v}$ ,  $V_H = 5 \text{v}$ ,  $V_L = 0 \text{v}$ ,  $U = 1 \text{v} \cdot \text{ns}^{-1}$ ,  $\mu_n C_{ox} = 25 \cdot 10^{-6} \text{A} \cdot \text{v}^{-2}$ . These are typical values to be found in the state-of-the-art circuit designs. From expression (8) we notice that the switch-induced error voltage of a NMOS switch is negative. In the following discussion we will focus on the absolute value of this error voltage and denote it as  $v_{dm}$ .

### A. Dependence on Gate Voltage Falling Rate

Error voltage,  $v_{dm}$ , is plotted against the gate voltage falling rate ranging from  $10^{-4} \text{ v.s}^{-1}$  to  $10^{-11} \text{ v.s}^{-1}$  for four signal voltage levels, 0v, 1v, 2v, and 3v in Fig. 3. In the case of slow falling rate, most of the channel charges return to the source when the switch is on, and the error voltage is primarily due to the clock feedthrough of the gate-drain overlap capacitance after the switch is turned off. At very slow falling rate,  $v_{dm}$  saturates at  $(V_S + V_T - V_L) C_{ol} / C_L$  as can be seen from expression (9). In the case of fast falling rate, nearly one half of the channel charges are deposited in the storage capacitor and  $v_{dm}$  saturates at  $V_{HT}(C_{ol} + C_{ox}/2)/C_L + (V_S + V_T - V_L) C_{ol} / C_L$  as can be seen from expression (10). From Fig. 3, it is obvious that the dependence of  $v_{dm}$  on  $V_S$  may be minimized by judiciously choosing the falling rate.

### B. Dependence on Signal Voltage Level and Substrate Doping

Error voltage,  $v_{dm}$ , is plotted against signal voltage  $V_S$  for five substrate dopings in Fig. 4. As the substrate doping increases, body effect increases accordingly, which in turn causes the threshold voltage in expression (8) to become more sensitive to  $V_S$ . The argument of the error function in expression (8) is smaller for larger  $V_S$  or heavier substrate doping. As long as the first term in (8) dominates, e.g. at fast falling rates,  $v_{dm}$  is a strong function of  $V_S$  and more so in the heavy-substrate-doping circuits.

### C. Dependence on Oxide Thickness

Advances in silicon technologies continue to make smaller MOS device dimensions possible. As the device size shrinks, the oxide thickness reduces, too. If storage capacitor oxide and gate oxide are scaled by the same factor,  $(C_{ol} + C_{ox}/2)/C_L$  remains constant. The effect of  $C_L$  increase due to oxide reduction is exactly balanced by the effect of  $\beta$  increase (assuming constant  $W/L$ ) such that the square root term and the error function term in expression (8)

are unaltered. Hence, error voltage  $v_{dm}$  is not affected.

#### D. Dependence on Channel Width and Length

Transistor size is one of the most important variables in circuit design. Designers have to choose the appropriate combination of transistor sizes in order to achieve optimum circuit performance. Error voltage  $v_{dm}$  is plotted against channel length ranging from  $1\ \mu\text{m}$  to  $10\ \mu\text{m}$  for four different channel widths,  $1\ \mu\text{m}$ ,  $4\ \mu\text{m}$ ,  $7\ \mu\text{m}$ , and  $10\ \mu\text{m}$  in Fig. 5. It is clear that smaller transistor size introduces smaller error voltage with the set of typical circuit parameter values listed at the beginning of this section.

#### E. Effect of Source Impedance

Source impedance of the signal voltage affects the error voltage. The circuit schematic which includes a source resistance is shown in Fig. 6. Derivation of the analytical model including source resistance is quite similar to that without source resistance and is attached as appendix B. Error voltage is plotted against source resistance in Fig. 7. As the source resistance increases, fewer channel charges return to the source end of the transistor and error voltage becomes larger.

### IV. Comparison with Computer Simulation

To validate the model, computer simulations using the SPICE 2G [5], [6] circuit-simulation program have been performed. The circuit configuration for computer simulations is the same as that of Fig. 1. One example of SPICE input file is as following:

SIMULATION OF SWITCH-INDUCED ERROR VOLTAGE ON A SWITCHED CAPACITOR

\*XQC<0.5 FOR CHARGE CONTROLLED MODEL

M1 2 1 3 0 MODN W=4U L=4U



- 8 -

```
CSTORAGE 2 0 2P
.MODEL MODN NMOS LEVEL=2 TOX=70N NSUB=5E14 KP=25U LD=0.35U
+VTO=0.6 VMAX=5E4 CGSO=1.7255E-10 CGDO=1.7255E-10 XQC=0.4999
VG 1 0 PWL(0 5 10N 5 60N 0)
VS 3 0 DC 0
.OPTIONS ABSTOL=1E-14 CHGTOL=1E-16 RELTOL=1E-5
.TRAN 1N 65N
.PRINT TRAN V(2) V(1)
.END
```

The analytical transient response, expression (6), and computer simulated results for three different gate voltage falling rates 0.1v/ns, 0.2v/ns, and 0.5v/ns are shown in Fig. 8. The close correspondence between the analytical analysis and the computer simulation is evident. The error voltage is plotted against the gate voltage falling rate for both the analytical and simulation results in Fig. 9. The agreement is excellent. Error voltages from both analytical and computer simulated results are shown in Fig. 7. Computer simulated result is shown in Fig. 7 together with the analytical result for the circuit schematic of Fig. 6. They match very well. Two other curves of simulated results in Fig. 7 correspond to the case where source capacitance exists in parallel with source resistance. The existence of the source capacitor compensates the effect of source resistance and inhibit error voltage from increasing too much. The larger the capacitance is, the more the compensation effect will be.

Charge controlled model is used in SPICE simulation by specifying the XQC parameter a value smaller than 0.5 so that charge conservation is retained. If XQC parameter is assigned a value greater than or equal to 0.5, Meyer's capacitance model is automatically employed and charge conservation is not guaranteed [7]. Meyer's capacitance model is implemented in SPICE MOS level-2

model in such a manner as to improve the convergence of circuit simulation [8]. However, it might introduce a small error to those transient simulations which are very sensitive to the capacitive currents of the transistors. The error introduced is insignificant for most circuit simulations but may be quite serious in simulating switched capacitor circuits. One curve corresponding to simulated results using SPICE MOS level-2 and Meyer's capacitance model is also shown in Fig. 9. SPICE simulation with Meyer's capacitance model generates an error of a fraction of a millivolt. The analytical model presented in this paper has no hidden error to the extent that its underlying assumptions, which are easily identified, are valid.

## V. Experimental Results

Experimental transistors for the MOS switch were designed and fabricated using a local oxidation polysilicon gate CMOS process. The transistor parameters are listed in Table I. The stray capacitance between the probes of a on-wafer testing station is quite large when the probes are close to each other. We put the transistors inside a 24-pin package to reduce such inter-probe capacitance.

Precision capacitance meter is employed to determine the effective storage capacitance,  $C_L$ , existing at the drain end. The measured value was 24.5pF. Fig. 10 shows a typical turn-off transient response of the switched capacitor circuit studied. The top curve is  $V_G(t)$  and the bottom curve is  $v_d(t)$ . The lower linear part of the bottom curve, corresponding to the second phase of switch turn-off, was used to extract the total capacitance between the gate pin and the drain pin of the switch, including the true transistor gate-drain overlap and the parasitic probe capacitances. The obtained value was 195fF. It was used as  $C_{ol}$  in

expression (8). Switch-induced error voltage was measured against the gate voltage falling rate ranging from  $4 \times 10^5 \text{ v} \cdot \text{s}^{-1}$  to  $3.3 \times 10^7 \text{ v} \cdot \text{s}^{-1}$  for two different signal voltage levels, 0v and 0.2v. Fig. 11 shows the measured data and calculated results from the analytical model (expression (8)). Good agreement is found in both cases.

## VI. Conclusion

An analytical expression for the switch-induced error voltage on a switched capacitor is presented. The expression plainly predicts the dependence of the error voltage on gate voltage falling rate, signal voltage level, transistor size, and process changes. For example, the error voltage increases with increasing  $V_S$  at low gate voltage falling rates and decreases with increasing  $V_S$  at high gate voltage falling rates. Computer simulations and experiments affirm the validity of the analysis. The compact expression (8) should be convenient in the analysis of switched capacitor circuits.

## Acknowledgement

The authors wish to thank Prof. Paul R. Gray and Prof. Don O. Pederson for their continuous encouragement and advice. Many helpful discussions from Cheng C. Shih, Ping W. Li, and Kyle W. Terrill are appreciated.

# Appendix A: Derivation of the Lumped Model from the Distributed Model

Referring to Fig. 12,

$$\begin{aligned}\frac{di}{dy} &= -C_{ox}' W \frac{d[V_G - V(y)]}{dt} \\ &\approx -C_{ox}' W \frac{dV_G}{dt}\end{aligned}\quad (A1)$$

$$i(y) = i(0) - C_{ox}' W \frac{dV_G}{dt} y \quad (A2)$$

$$\begin{aligned}i_d = i(L) &= i(0) - C_{ox}' W L \frac{dV_G}{dt} \\ &= i(0) - C_{ox} \frac{dV_G}{dt}\end{aligned}\quad (A3)$$

$$\begin{aligned}\frac{dv}{dy} &= i(y) \frac{R_{ch}'}{W} \\ &= i(0) \frac{R_{ch}'}{W} - C_{ox}' R_{ch}' \frac{dV_G}{dt} y\end{aligned}$$

$$v(y) = i(0) \frac{R_{ch}'}{W} y - C_{ox}' R_{ch}' \frac{dV_G}{dt} \frac{y^2}{2} \quad (A4)$$

$$\begin{aligned}v_d = v(L) &= i(0) \frac{R_{ch}'}{W} L - C_{ox}' R_{ch}' \frac{dV_G}{dt} \frac{L^2}{2} \\ &= \frac{i(0)}{G} - \frac{C_{ox}}{2G} \frac{dV_G}{dt}\end{aligned}\quad (A5)$$

Eliminate  $i(0)$  from (A3) by using (A5)

$$i_d = v_d G - \frac{C_{ox}}{2} \frac{dV_G}{dt} \quad (A6)$$

It is obvious that

$$C_L \frac{dv_d}{dt} = -i_d + C_{ol} \frac{dV_G}{dt} \quad (A7)$$

Hence we obtain the desired expression,

$$C_L \frac{dv_d}{dt} = -G v_d + \left( C_{ol} + \frac{C_{ox}}{2} \right) \frac{dV_G}{dt}$$

- 12 -

$$= -i_d + \left( C_{o1} + \frac{C_{ox}}{2} \right) \frac{dV_G}{dt} \quad (A8)$$

Expression (A8) may be interpreted with a equivalent lumped circuit as shown in

Fig. 2(a). It is trivial to show that Fig. 12 reduces to Fig. 2(b) for  $V_G < V_S + V_T$ .

- 13 -

### Appendix B: Derivation of the Analytical Model Including Source Resistance

Referring to Fig. 8 and Expression (A8), KCL law at node A and node B require

$$C_L \frac{dv_d}{dt} = -i_d + (C_{ol} + \frac{C_{ox}}{2}) \frac{d(V_G - v_d)}{dt} \quad (B1)$$

$$\frac{v_s}{R_S} = i_d + (C_{ol} + \frac{C_{ox}}{2}) \frac{d(V_G - v_s)}{dt} \quad (B2)$$

With the same assumptions in section II, when the transistor is on, (B1) and (B2) simplify to

$$C_L \frac{dv_d}{dt} = -\beta(V_{HT} - Ut)(v_d - v_s) - (C_{ol} + \frac{C_{ox}}{2})U \quad (B3)$$

$$\frac{v_s}{R_S} = \beta(V_{HT} - Ut)(v_d - v_s) + (C_{ol} + \frac{C_{ox}}{2})U \quad (B4)$$

Using (B4) to eliminate  $v_s$  from (B3), we obtain a first order differential equation,

$$C_L \frac{dv_d}{dt} = - \frac{\beta(V_{HT} - Ut)}{1 + \beta R_S(V_{HT} - Ut)} v_d - (C_{ol} + \frac{C_{ox}}{2}) \left[ 2 - \frac{1}{1 + \beta R_S(V_{HT} - Ut)} \right] U \quad (B5)$$

Solving this differential equation and including the clock feedthrough due to gate-drain overlap capacitance when the transistor is off, we get the complete solution,

$$v_{dm} = \frac{C_{ol}}{C_L} (V_S + V_T - V_L) + U \left[ \frac{C_{ol} + \frac{C_{ox}}{2}}{C_L} \right] \exp \left[ - \frac{V_{HT}}{U C_L R_S} \right] \\ \times \int_0^{\frac{V_{HT}}{U}} \left[ \beta R_S (V_{HT} - U \xi) + 1 \right] \frac{1}{C_L \beta R_S \cdot R_S U} \exp \left[ \frac{\xi}{C_L R_S} \right] \left[ 2 - \frac{1}{1 + \beta R_S (V_{HT} - U \xi)} \right] d\xi \quad (B6)$$

- 14 -

TABLE I

NMOS SWITCH PARAMETERS	
$V_T (V_{SB}=0.0v)$	0.70v
$V_T (V_{SB}=0.2v)$	0.81v
$W_{DRAWN}$	$40\mu m$
$L_{DRAWN}$	$8\mu m$
$L_D$	$0.45\mu m$
$\beta$	$295\mu A \cdot v^{-2}$
$t_{ox}$	85nm



### Figure Captions

- Fig. 1. Schematic of the switch circuit under study.
- Fig. 2. Equivalent lumped models for the circuit shown in Fig. 1.  
(a)  $V_H \geq V_G \geq V_S + V_T$  (b)  $V_S + V_T \geq V_G \geq V_L$ . This equivalent circuit is derived in appendix A.
- Fig. 3. The error voltage as a function of the gate voltage falling rate for four signal voltage levels.
- Fig. 4. The error voltage as a function of the signal voltage level for five substrate dopings.
- Fig. 5. The error voltage as a function of transistor channel length for four channel widths.
- Fig. 6. Schematic of the switch circuit with source resistor.
- Fig. 7. Comparison of the analytic and computer simulated results of the error voltage as a function of source resistance. Computer simulated results with 0.5pF/1pF capacitance in parallel with  $R_S$  are also shown.
- Fig. 8. Comparison of the analytic and computer simulated transient responses for three different gate voltage falling rates.
- Fig. 9. Comparison of the analytic and computer simulated results of the error voltage as a function of the gate voltage falling rate.
- Fig. 10. Turn-off transient response of the switched capacitor circuit shown in Fig. 1. The top curve is the waveform applied to the gate. The bottom curve is the error voltage waveform at the drain.
- Fig. 11. Measured and calculated error voltages as functions of gate voltage falling rate for two signal voltage levels.
- Fig. 12. Distributed model for the circuit shown in Fig. 1.

# REFERENCES

- [1] P. R. Gray, D. A. Hodges, and R. W. Brodersen, (Eds.), Analog MOS Integrated Circuits. New York: IEEE Press, 1980.
- [2] R. E. Suarez, P. R. Gray, D. A. Hodges, "All-MOS Charge Redistribution Analog-to-Digital Conversion Techniques: Part II," IEEE J. Solid-State Circuits, vol. SC-10, pp. 379-385, Dec. 1975.
- [3] R. C. Yen, P. R. Gray, "A MOS Switched-Capacitor Instrumentation Amplifier," IEEE J. Solid-State Circuits, vol. SC-17, pp. 1008-1013, Dec. 1982.
- [4] D. Schmitt, G. Dorda, "Interface States in MOSFETs due to Hot-Electron Injection Determined by the Charge Pumping Technique," Electronics Letters, vol. 17, no. 20, Oct. 1981.
- [5] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Electron. Res. Lab., Univ. California, Berkeley, memo. ERL-M520, May 1975.
- [6] A. Vladimirescu, S. Liu, "The simulation of MOS Integrated Circuits Using SPICE2," Electron. Res. Lab., Univ. California, Berkeley, memo. ERL-M80/7, Oct. 1980.
- [7] D. E. Ward, R. W. Dutton, "A Charge-Oriented Model for MOS Transistor Capacitances," IEEE J. Solid-State Circuits, vol. SC-13, pp. 703-708, Oct. 1978.
- [8] A. Vladimirescu, private communication.

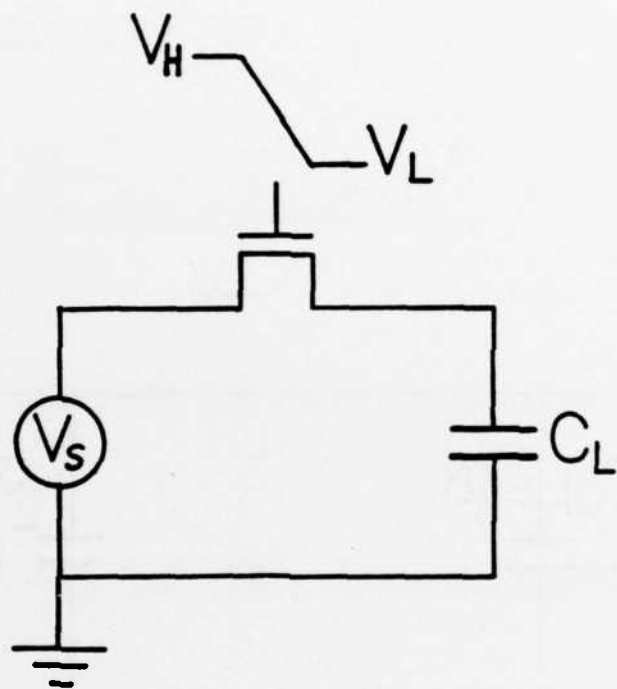


Fig. 1

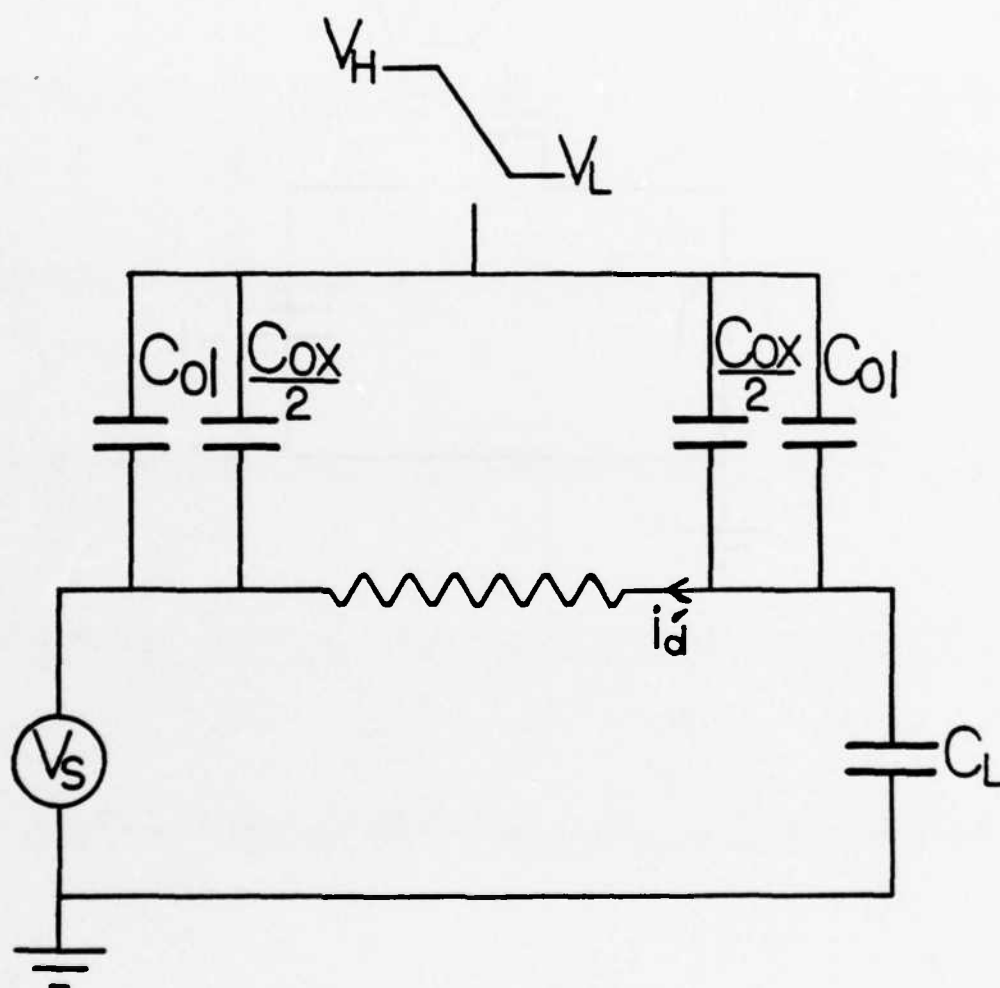


Fig. 2(c)

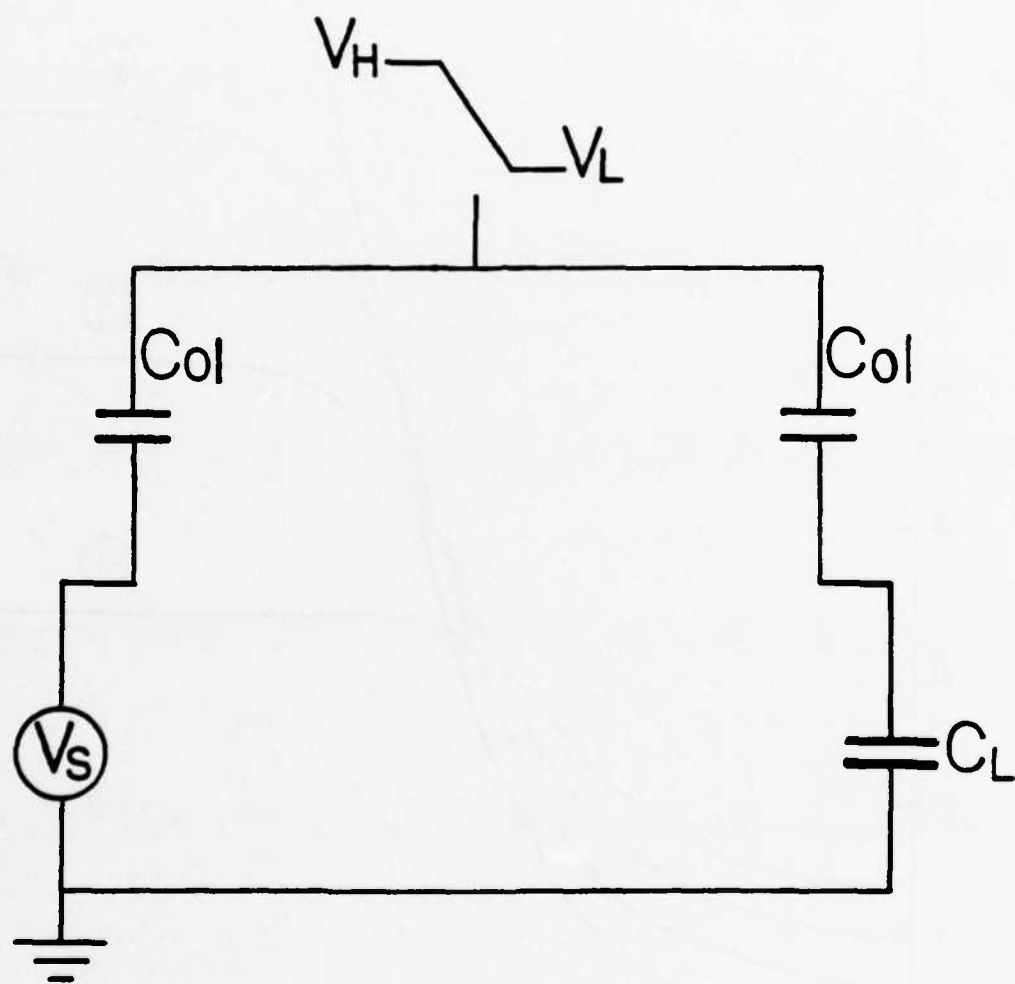
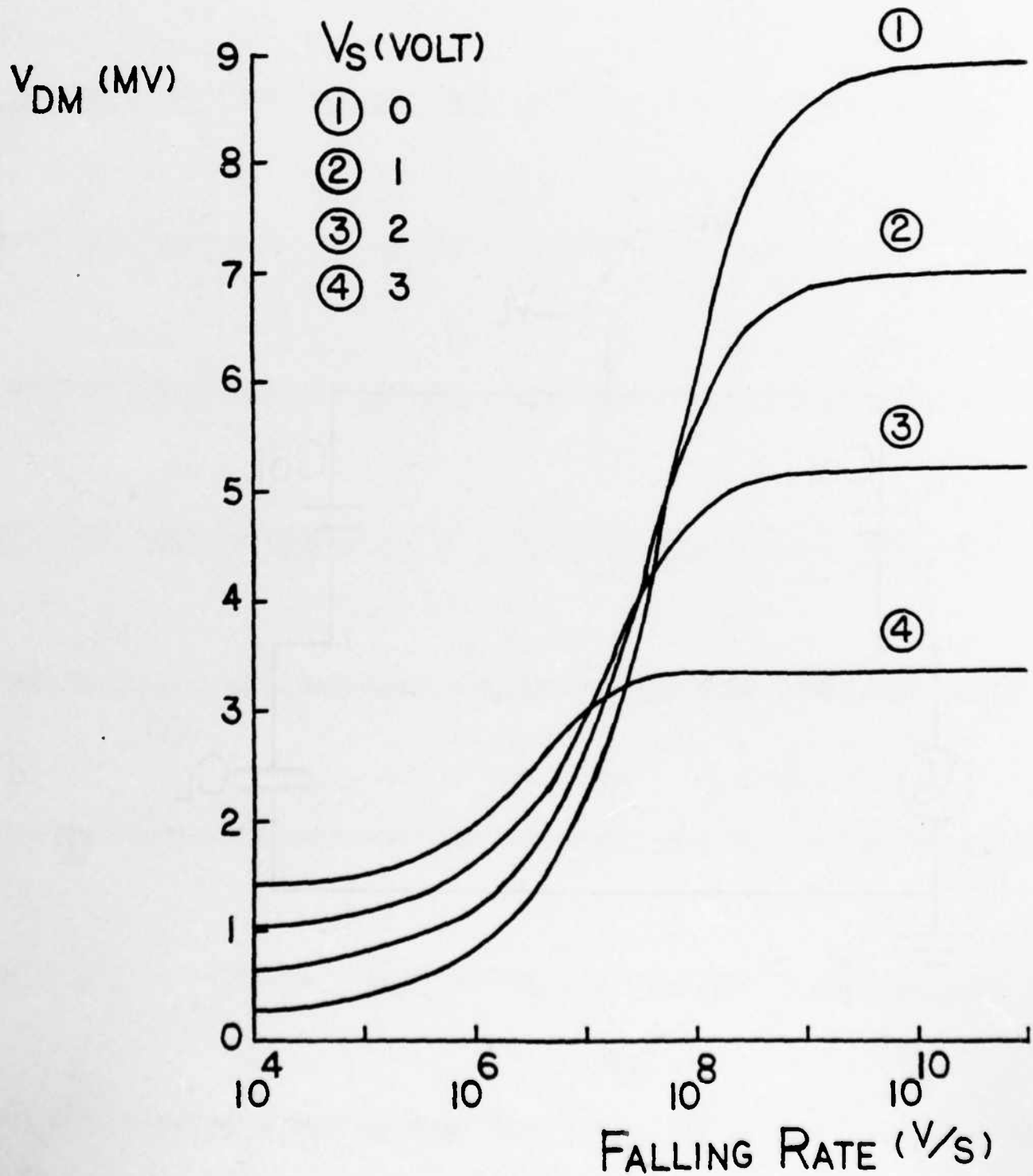
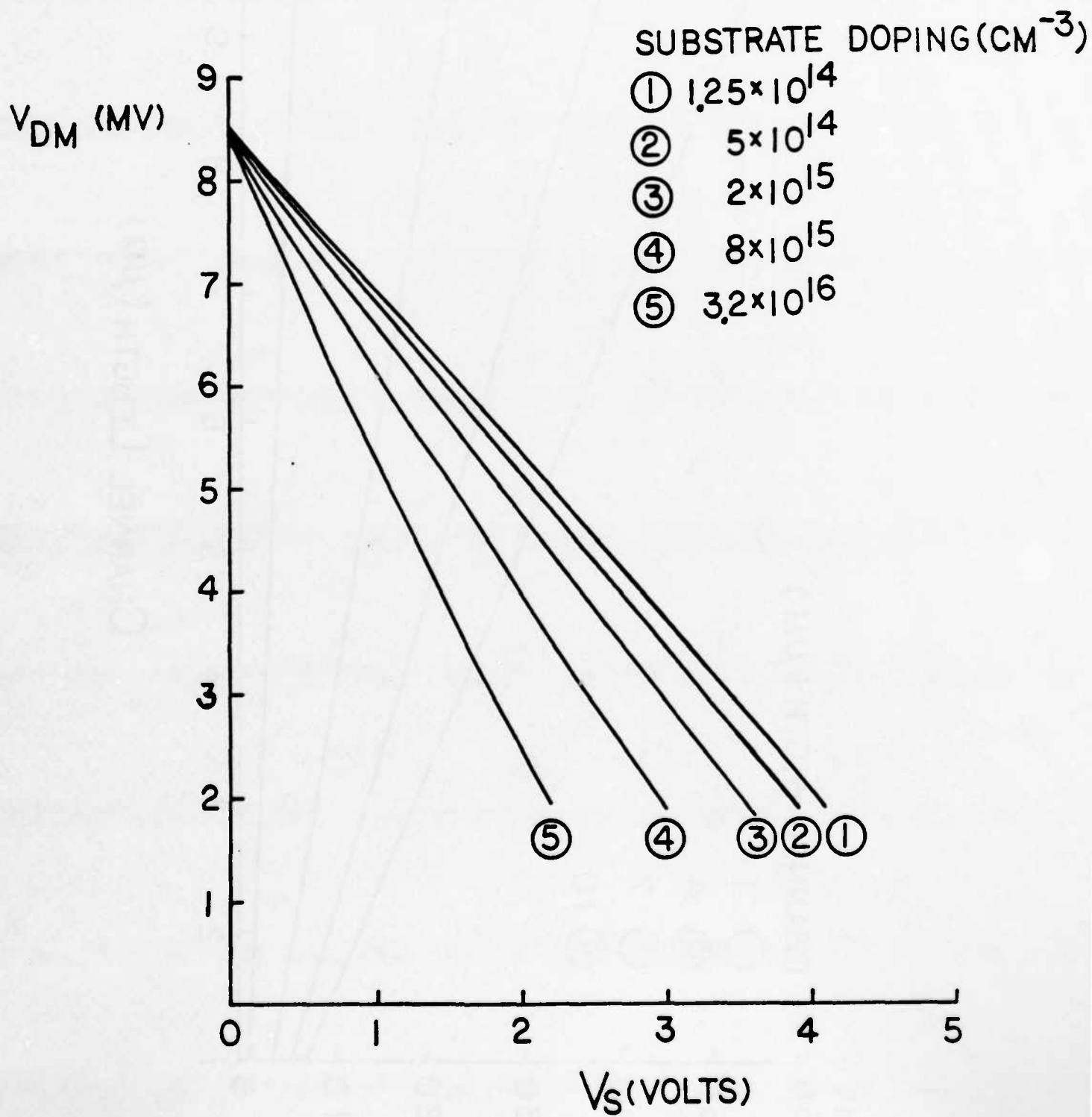
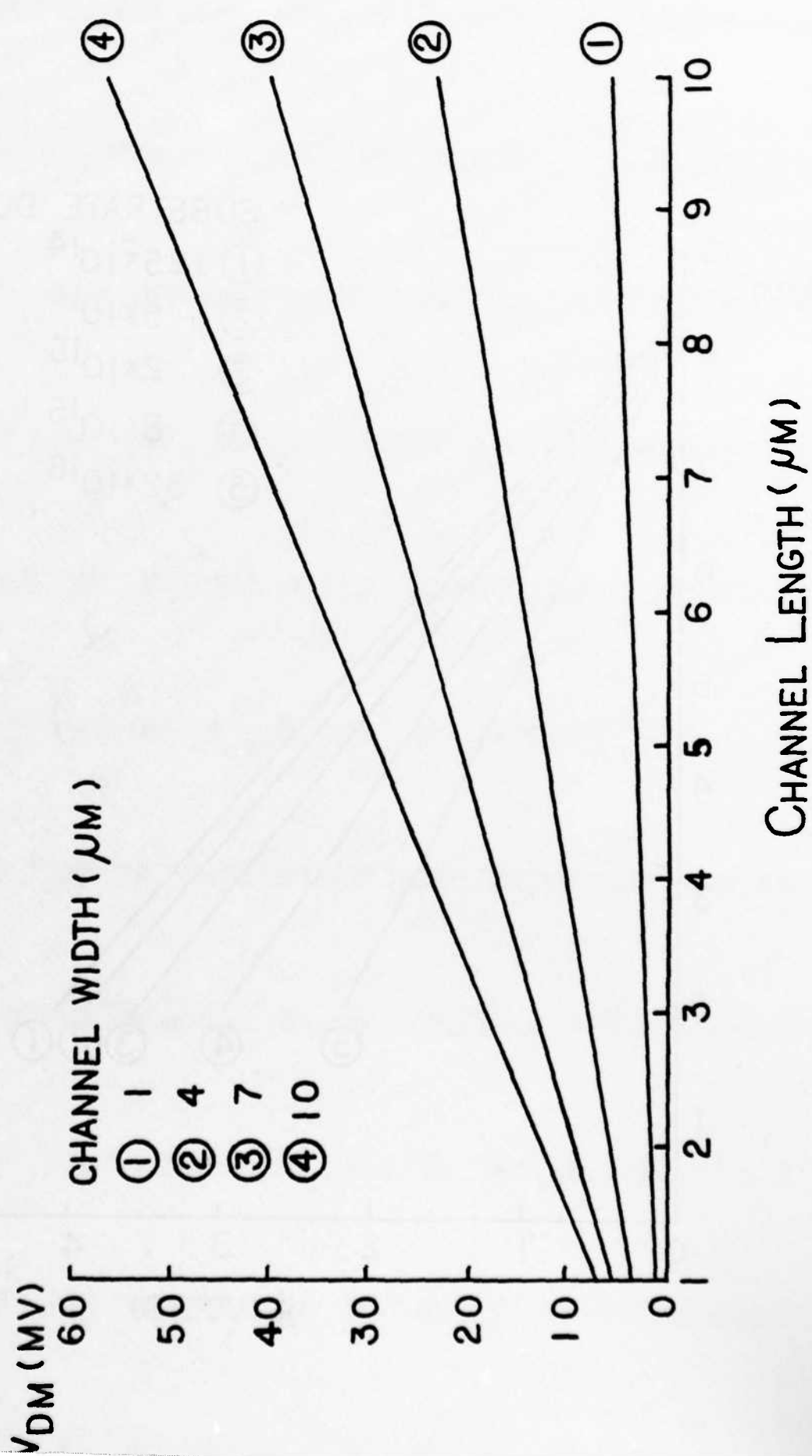


Fig. 2(b)









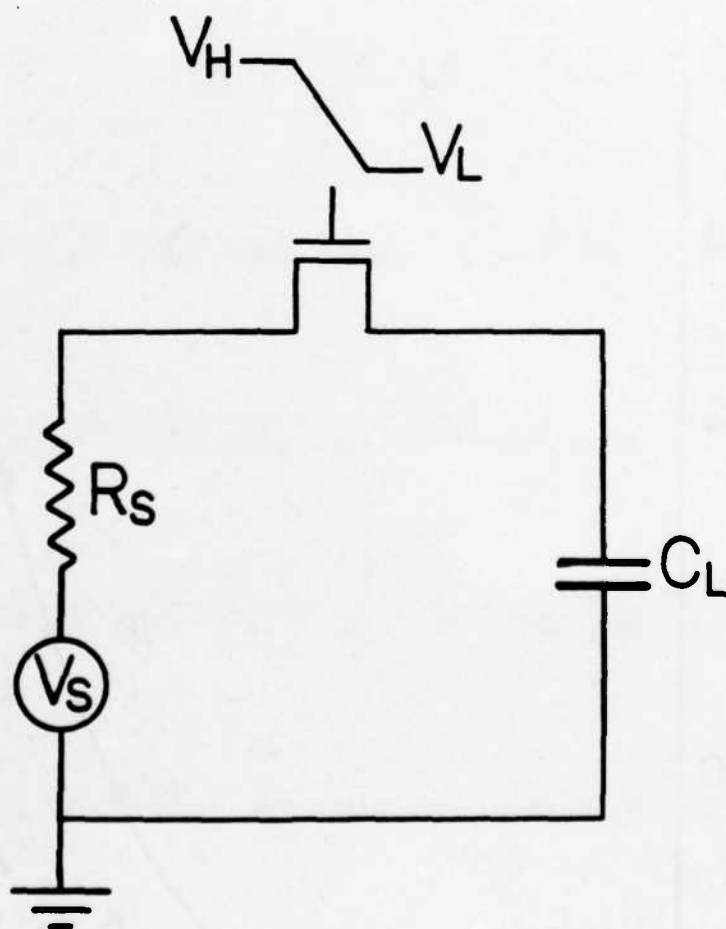


Fig 6

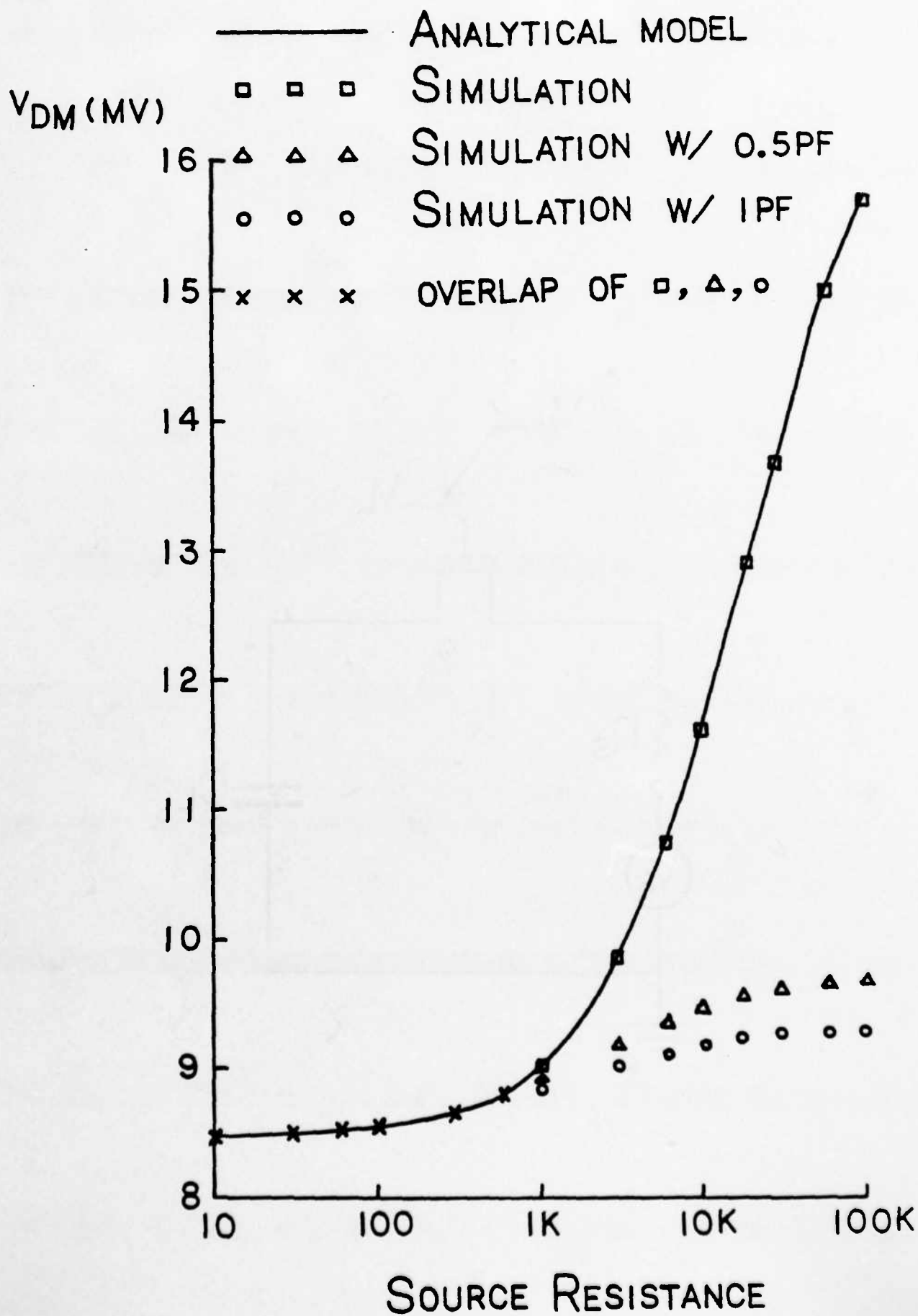


Fig. 7

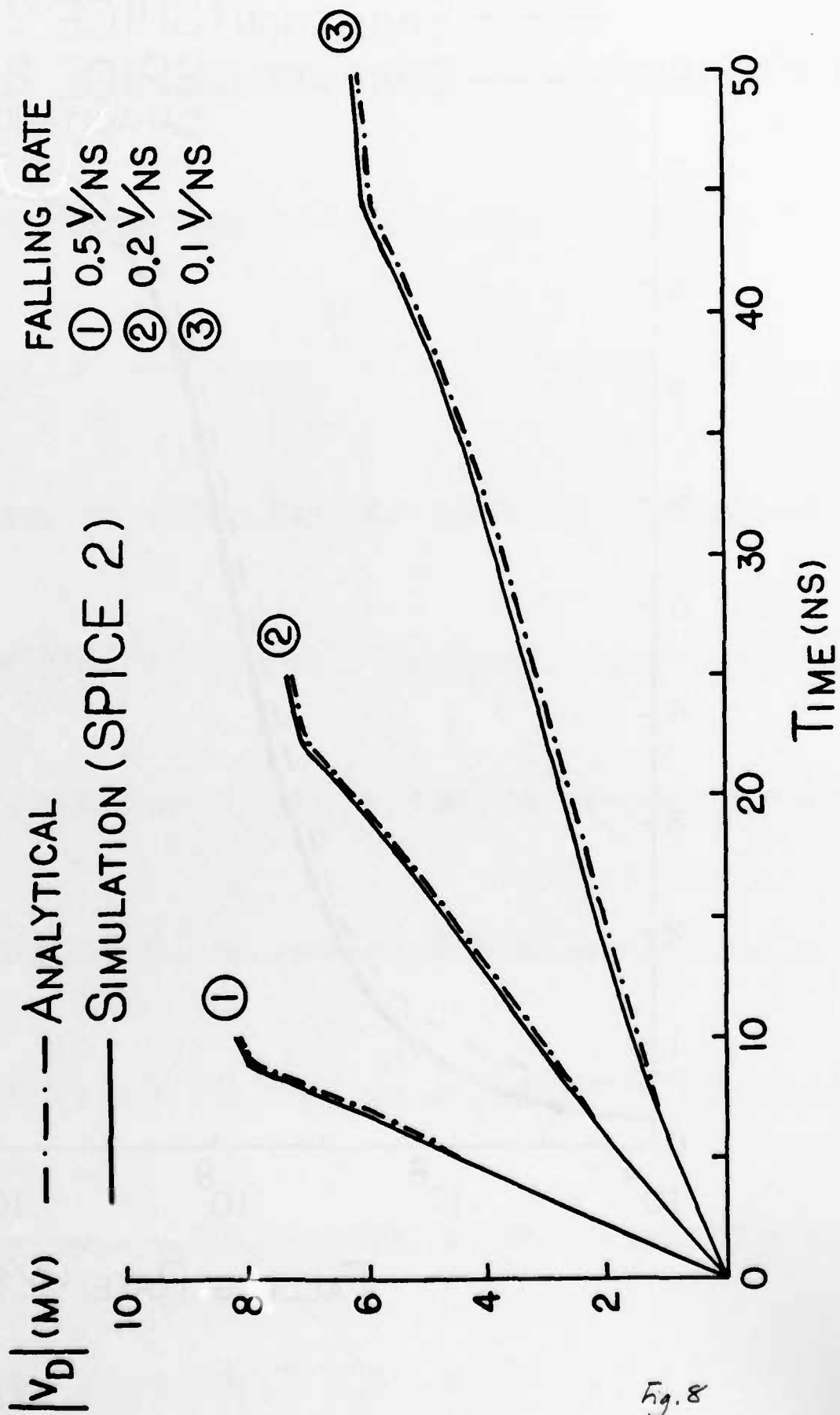


Fig. 8

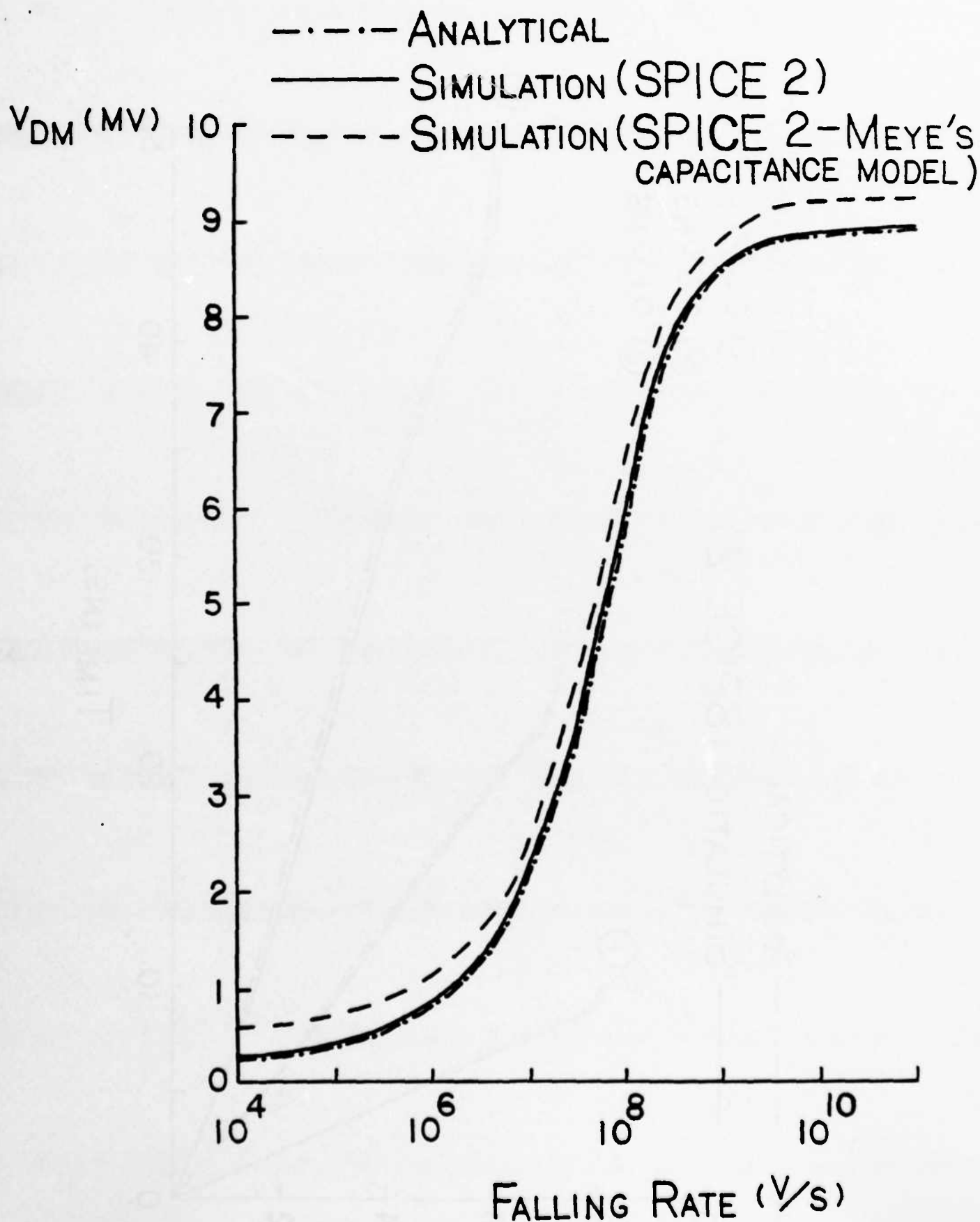
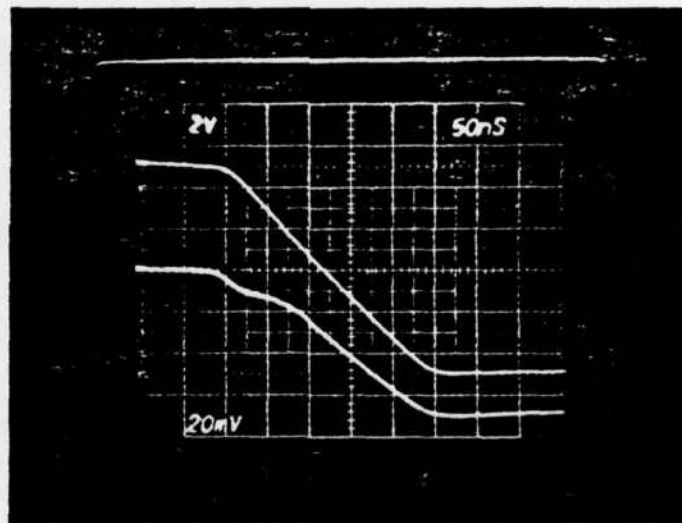


Fig. 9



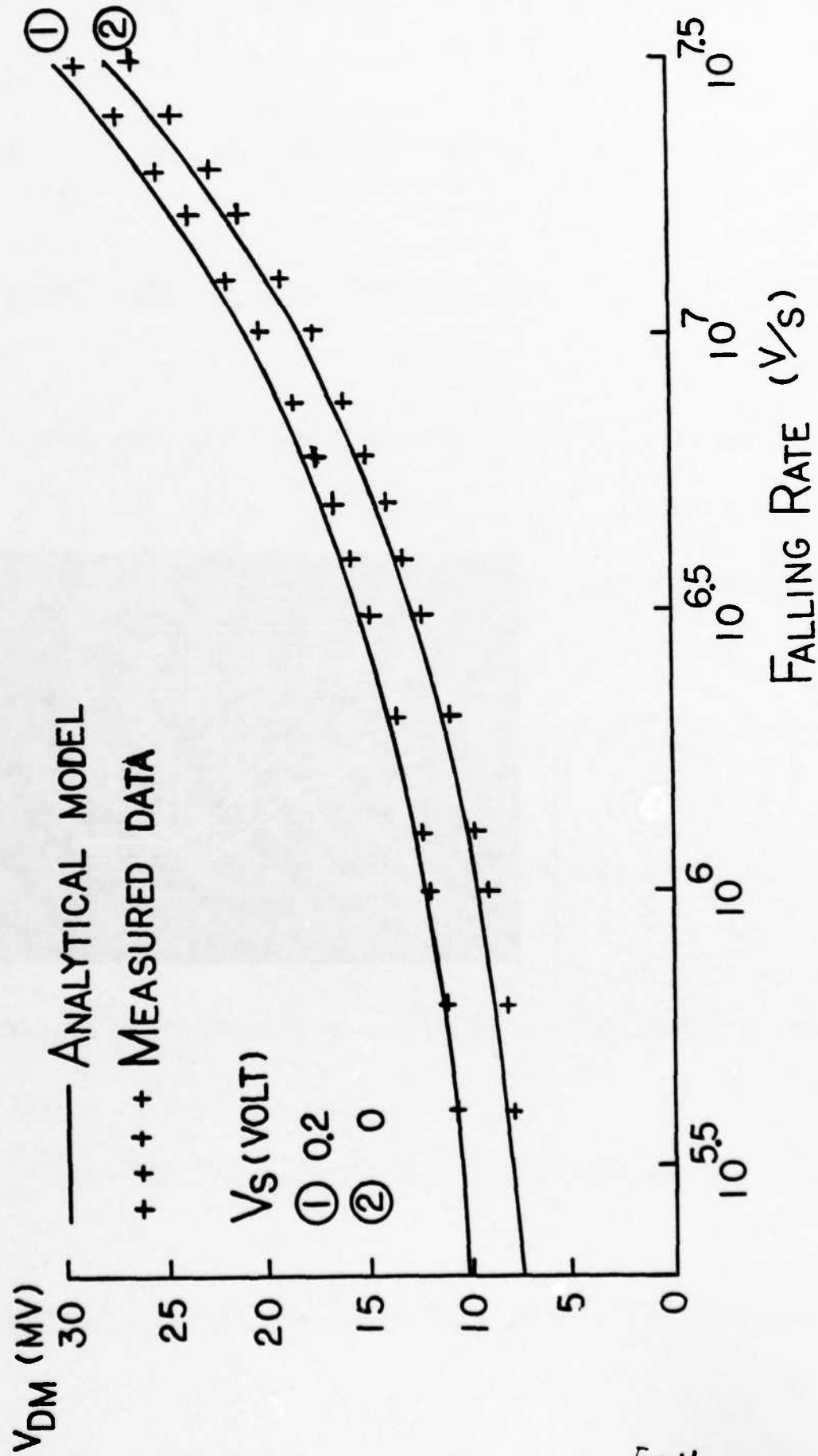


Fig. 11



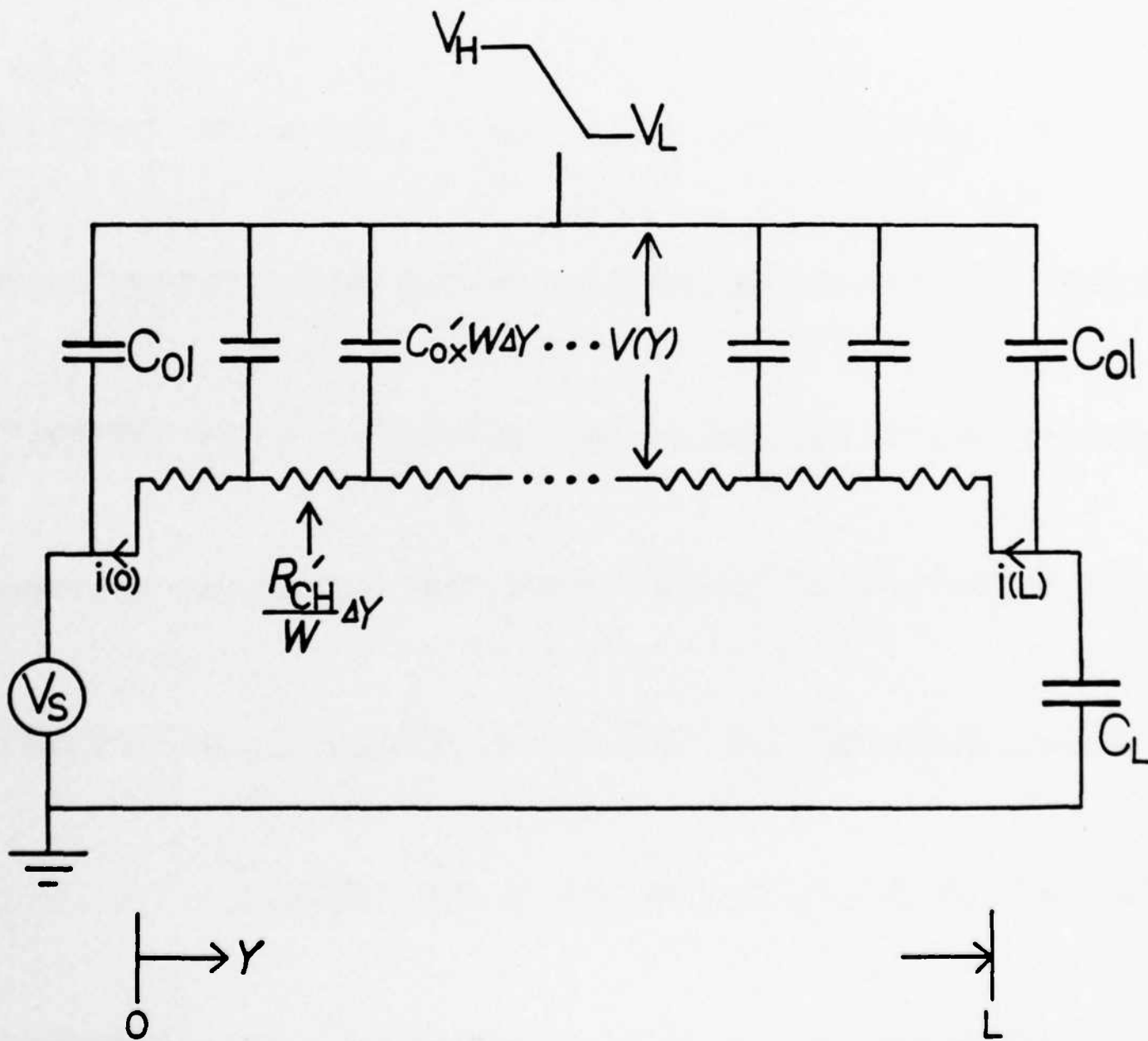


Fig. 12

## SUBSTRATE POTENTIAL CALCULATION FOR LATCH-UP MODELING

*K. TERRILL and C. HU*

Department of Electrical Engineering and Computer  
Sciences  
University of California, Berkeley

### *ABSTRACT*

The input and output circuits are the main triggering mechanisms of latch-up in CMOS technology. We have studied the triggering capability of these circuits and the effectiveness of using guard-rings to suppress triggering. We present a method to estimate the substrate potential induced by a triggering current in these circuits and the effect of using guard-rings to prevent latch-up. It was necessary to include effects of the field-threshold implant to obtain good agreement between theory and measurements.

## SUBSTRATE POTENTIAL CALCULATION FOR LATCH-UP MODELING

*K. TERRILL and C. HU*

Department of Electrical Engineering and Computer  
Sciences  
University of California, Berkeley

### INTRODUCTION

Bulk CMOS integrated circuits contain both parasitic vertical and lateral bipolar transistors. These transistors form a pnpn structure (a generic SCR) that can latch-up if the positive feedback between the coupled transistors produces regenerative switching.<sup>1</sup> It is possible to prevent latch-up by suppressing those mechanisms which trigger the SCR into the turned-on state. The main triggering mechanisms discussed in the literature are current injection from the input and/or output circuitry. Either majority or minority carrier current can be injected into the substrate from the I/O circuitry. We investigate in this study the possible triggering due to majority carrier current and also look at the effectiveness of using guard rings to prevent triggering for this case. Although circuit models for the parasitic SCR have been presented<sup>2,3</sup> these models give no means for calculating the substrate resistance. The method we present allows the calculation of this resistance.

Fig. 1 shows a typical input protection circuit used in an N well technology.<sup>4</sup> When the input voltage is positively large the forward biased p+/n diode injects minority carriers into the well after which they are quickly swept into the substrate and become majority carriers. In the substrate these carriers generate a three dimensional ohmic drop as they spread-out. If the current is high then the surface potential may become large enough to forward bias a nearby diode.

The minority carriers injected by this nearby diode that are collected by a well may lead to latch-up. A grounded guard ring located either near the injector or the diode will help hold the surface potential at ground thus preventing the diode from becoming forward biased. It is also possible for the drain of an output device to become forward biased inside the well. The forward biasing of a drain p+/n junction will cause the same effect as described above. Although we present the case of an N well technology, the above arguments also hold for a P well technology with a simple sign reversal for voltage and current. Another possible mechanism for substrate current is Hot-Electron generated substrate currents from n MOS devices.

We are using p+ surface contacts to emulate the injection of current into the substrate from a forward biased p+/n junction inside the well. Measurements on a CMOS N well technology wafer have shown that the surface potential due to a forward biased diode inside does not differ by more than 15% from the surface potential due to a surface contact when both are injecting the same current. There was no significant variation in the surface potential when the N well bias voltage was varied as long as the well/substrate junction was reversed biased. During all measurements the backside is grounded. In what follows we will refer to the ratio of the surface potential  $\phi_s(r)$  to the injecting current as the transverse resistance  $R_t$  at point r.

## DISCUSSION

A calculation of the surface potential due to a surface contact must take into account the highly conductive field implantation layer on the silicon surface. This layer typically is about one micron in depth and has a resistivity which is at least an order of magnitude lower than the substrate resistivity. The main effect of this low resistivity surface layer is the reduction in the spreading

resistance between a surface contact and the backside contact. If we assume this low resistivity region has a uniform concentration the boundary conditions corresponding to this case are:

- I The potential at the surface in the contact region is constant.
- II The current component at and normal to the surface is zero in the noncontacted region.
- III Both the potential and the current are continuous across the boundary separating the low resistivity surface region from the substrate.
- IV The potential at the back of the wafer is zero.

If the region in which the surface potential is to be calculated is small compared to the thickness of the wafer then the fourth boundary condition can be replaced with an infinite substrate with little variation in the calculated potential. We have assumed an infinite substrate in all calculations that follow. The wafers utilized have a thickness of ~325 micron and the results of our calculations are indeed accurate for lateral distances less than this thickness.

It has been shown that the potential distribution due to a surface contact, on a substrate composed of a thin, low-resistivity layer overlying a high resistivity substrate, is not strongly influenced by the particular form of an assumed injecting current distribution.<sup>5</sup> We will utilize this fact to find an approximate closed form solution for the case of a circular contact. This is true as long as the dimensions of the surface contact are not less than the depth of the conductive layer.

## RESULTS

The resistance between a square surface contact of side-length  $L$  and the backside is shown in Fig. 2 as a function of both field implant resistivity and

side-length. The resistance is calculated by using a circular disk of the same area and assuming the current on the disk has the same distribution as the case of a disk on a homogeneous substrate <sup>6</sup>. By using this assumed current distribution the surface potential can be found by imaging this current through both the surface boundary and the boundary between the low resistivity layer and the substrate. The field implant profile is assumed to be of uniform concentration with a depth equal to 1 micron and the substrate resistance is 21 ohm-cm. The resistance between the surface contact and the backside is given by:

$$R = \frac{\rho_f}{2\pi a} \left\{ \frac{\pi}{2} + 2 \sum_{n=1}^{\infty} \kappa^n \sin^{-1} \left( \frac{2a}{\sqrt{(2a)^2 + (2nT_f)^2 + 2nT_f}} \right) \right\} \quad (1)$$

Where :

$$\kappa = \frac{\rho_s - \rho_f}{\rho_s + \rho_f} \quad ; \quad a = \sqrt{\frac{L^2}{\pi}}$$

$\rho_s$  is the substrate resistivity,  $\rho_f$  is the resistivity of the field implant region and  $T_f$  is the depth of this region below the surface. The correlation between the calculated and the measured resistance are as close as can be expected since the actual field implant profile can not be accurately determined. The field resistivity used to fit the data differed by 11% from the average resistivity SUPREM II predicted over this region and the substrate resistance was obtained from a four point probe measurement.

Fig. 3 shows both the measured and calculated transverse resistance of a surface contact as a function of the distance from the edge of the injector "d" and the injector side-length "L". We define the transverse resistance at a distance d to be the ratio of the surface potential at that distance to the injector current. Here the field resistivity is fixed at 1.20 ohm-cm. The transverse resistance is given by:

$$R_t = \frac{\rho_f}{2\pi a} \left\{ \sin^{-1} \left( \frac{a}{(d+a)} \right) + \right. \quad (2)$$

$$\left. 2 \sum_{n=1}^{\infty} \kappa^n \sin^{-1} \left( \frac{2a}{\sqrt{d^2 + (2nT_f)^2} + \sqrt{(d+2a)^2 + (2nT_f)^2}} \right) \right\}$$

We also present in Fig. 3 the solution of a surface contact on a homogeneous substrate with  $L$  equal to 10 micron for comparison. As the distance  $d$  increases the transverse resistance asymptotically approaches the value predicted for a homogeneous substrate.

Fig. 4 shows the transverse resistance and resistance to backside for a surface contact which completely encloses a region in its interior. The experimental geometry used is also shown in Fig. 4. The experimental geometry is a square loop where each side is of length  $L$  (50 micron) and thickness  $T$  (variable). In order to simplify the calculation the enclosing guard-ring is assumed to be a circular ring of inner radius  $r_a$  and outermost radius  $r_b$ . Since no analytical solution exist for this geometry it was necessary to solve for the potential by using numerical methods. The potential was assumed constant over the injecting region since this is the proper boundary condition for a surface contact. The current density through a circular, enclosing guard-ring, which has a constant potential  $V$ , can be formulated as an integral equation.

$$V = \int_{r_a}^{r_b} r' dr' J(r') G(r, r') \quad (3)$$

Here  $r$  is restricted to the injecting region where  $\phi_s = V$ .

$$G(r, r') = \rho_f \int_0^{\infty} dq J_0(qr') J_0(qr) \left\{ \frac{1 + \kappa e^{-2qT_f}}{1 - \kappa e^{-2qT_f}} \right\} \quad (4)$$



The integral equation is solved numerically to obtain the current density  $J(r')$  between  $r_a$  and  $r_b$ . After the current density is obtained the surface potential can be computed:

$$\varphi(r) = \int_{r_a}^{r_b} r' dr' J(r') G(r, r') \quad (5)$$

The circular ring used is that with the same total area as the test structure and a mean radius equal to  $(L+T)/2$ . It is important that ample contacts be made to the guard-ring structure to maintain a constant potential over the guard-ring. The improvement in completely enclosing a diode is clearly displayed here where the surface potential drops much less rapidly inside the guard-ring than outside the guard-ring. Although all the data is given for a substrate resistivity of 21 ohm-cm the values of the resistances can be scaled with the substrate resistivity as long as the ratio between the field resistivity and the substrate resistivity is held fixed.

The solution for an injector in the presence of a guard-ring can be found once the surface contact to backside resistance and the transverse resistances are determined. Before showing this we must first define some terms that are necessary in our formulation.

The terms are defined as follows:

$$R_{di} = \frac{V_d}{I_i} \text{ with } I_g = 0 \text{ and } I_d = 0$$

$$R_{dg} = \frac{V_d}{I_g} \text{ with } I_i = 0 \text{ and } I_d = 0$$

$$R_{ig} = \frac{V_i}{I_g} \text{ with } I_i = 0 \text{ and } I_d = 0$$

$$R_{gi} = \frac{V_g}{I_i} \text{ with } I_g=0 \text{ and } I_d=0$$

$$R_g = \frac{V_g}{I_g} \text{ with } I_i=0 \text{ and } I_d=0$$

Where "i", "g" and "d" refer to the injector the guard ring and the nearby diode respectively.  $R_{di}$ ,  $R_{dg}$ , ect are transverse resistances.  $R_g$  is the resistance between the guard-ring and the backside. All the terms except the last may be calculated by equation (2) or (5). It is necessary when doing so to average the potential given by these equations over the region of interest. The last term, the resistance between the guard-ring and the backside, maybe calculated using equations (1) or (3).

Since the current injected into the substrate by the forward biased diode inside the well is a constant current (independent of substrate potential), a solution can be found by simple superposition of the potential due to the injecting source and the potential due to the guard-ring. If the guard-ring is grounded then the guard-ring current must be:

$$I_g = \frac{R_{gi}}{R_g} I_i \quad (6)$$

It is important to note that the substrate current is equal to the difference between the injected current and the guard-ring current. This current path cannot be neglected unless  $R_{gi}$  is almost equal to  $R_g$ . From Figs. (2) and (3) it is easy to see that this condition does not exist since the transverse resistance drops rapidly as the distance from the injector edge increases.

The solution for an injector in the presence of a guard-ring is given by:

$$\frac{V_d}{I_i} = R_{di} - \frac{R_{dg} R_{gi}}{R_g} \quad (7)$$

From reciprocity considerations it is easy to show that  $R_{iq} = R_{gi}$ , ect. This shows that the same transverse resistance is obtained if the injector and the nearby diode are switched so that  $i \Rightarrow d$  and  $d \Rightarrow i$  in the above equations. When calculating  $R_{di}$  and  $R_{gi}$  we did not account for the constant potential, guard-ring surface contact. Although this contact is floating it still has some effect on the potential. In neglecting this floating guard-ring we introduce some error in our calculation.

As an example in Fig. 5 we show the effective substrate resistance ( $V_d / I_i$ ) due to a injector of side-length  $L=12$  micron which has a nearby guard-ring of side-length  $L=12$  micron located at a distance  $d=18$  micron from the edge of the injector. We present two cases of interest. In the first case the guard-ring is located between the injector and the diode while in the second case the guard-ring is placed on the opposite side of the injector. For both cases  $R_g = 2.75 K \text{ ohm}$  and  $R_{ig} = .971 K \text{ ohm}$  using equations (1) and (2) respectively. The transverse resistance  $R_{ig}$  was average over the region of interest. The guard-ring current is, thus, equal to 35.3 % of the injected current. The remaining current passes through the backside contact. We also show the resistance due to an injector with no guard-ring present for comparison as a broken line. As expected the placement of the guard-ring has a major impact on its ability to protect a nearby diode. When the guard-ring is placed between the injector and the diode its effectiveness is improved, so that the substrate resistance is reduced considerably.

## CONCLUSIONS

We have presented a means to calculate the substrate potential induced by a triggering current as in typical CMOS I/O circuitry. This method may also be used to find the substrate resistance in internal latch-up structures. The

effectiveness of using a guard-ring to suppress triggering can be computed by using superposition. In order to obtain an accurate prediction it was necessary to account for the effect of the field-threshold implant. It is important to include the current path to the backside when estimating the substrate resistance in latch-up structures.

#### **ACKNOWLEDGEMENT**

This research was sponsored by the Defense Advanced Research Projects Agency contract N00039-K-0251.

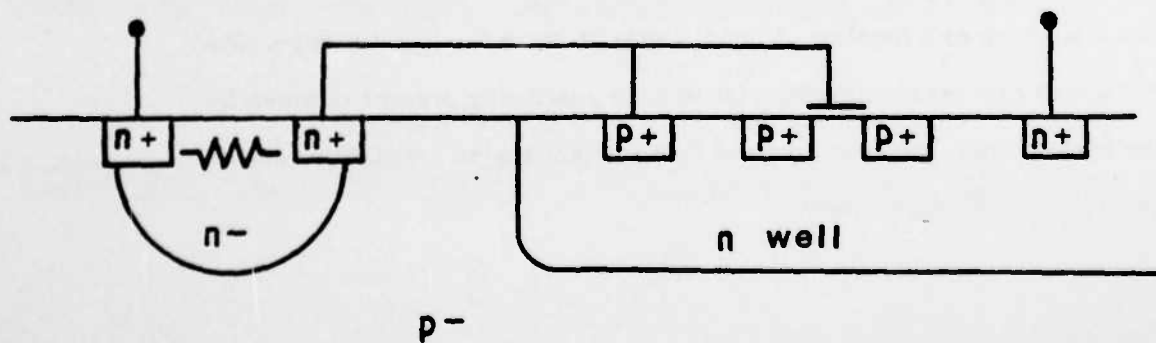
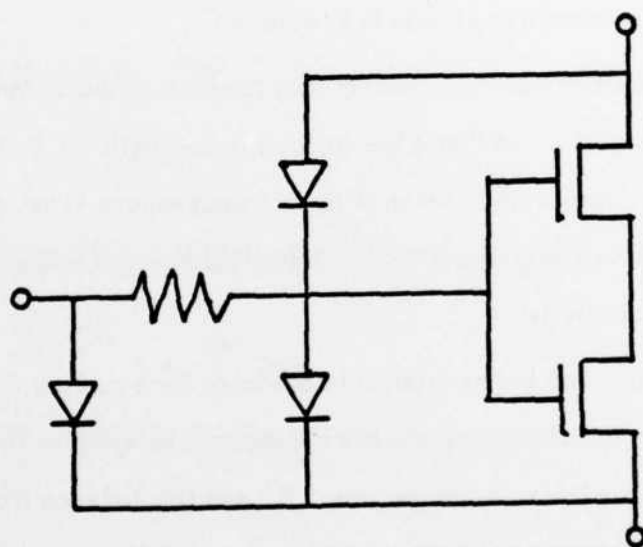
REFERENCES

- 1] B. L. Gregory and B. D. Shafer IEEE Trans. Nuc. Sci. NS20, pp 293 1973.
- 2] J. E. Schroeder, A. Ochoa, Jr. and P. V. Dressendorfer IEEE Trans. Nuc. Sci. NS27, pp 1735 1980.
- 3] W. D. Raburn IEEE International Electron Devices Meeting, pp 252 1980.
- 4] Estreich, D. B. , Ph. D. Dissertation, Stanford University, 1980.
- 5] M. S. Leong, S. C. Choo and L. S. Tan Solid-State Electronics Vol. 25. No 9, pp 877, 1982.
- 6] Walter David Jackson, Classical Electrodynamics, third printing pp. 92. John Wiley & Sons, Inc., New York London

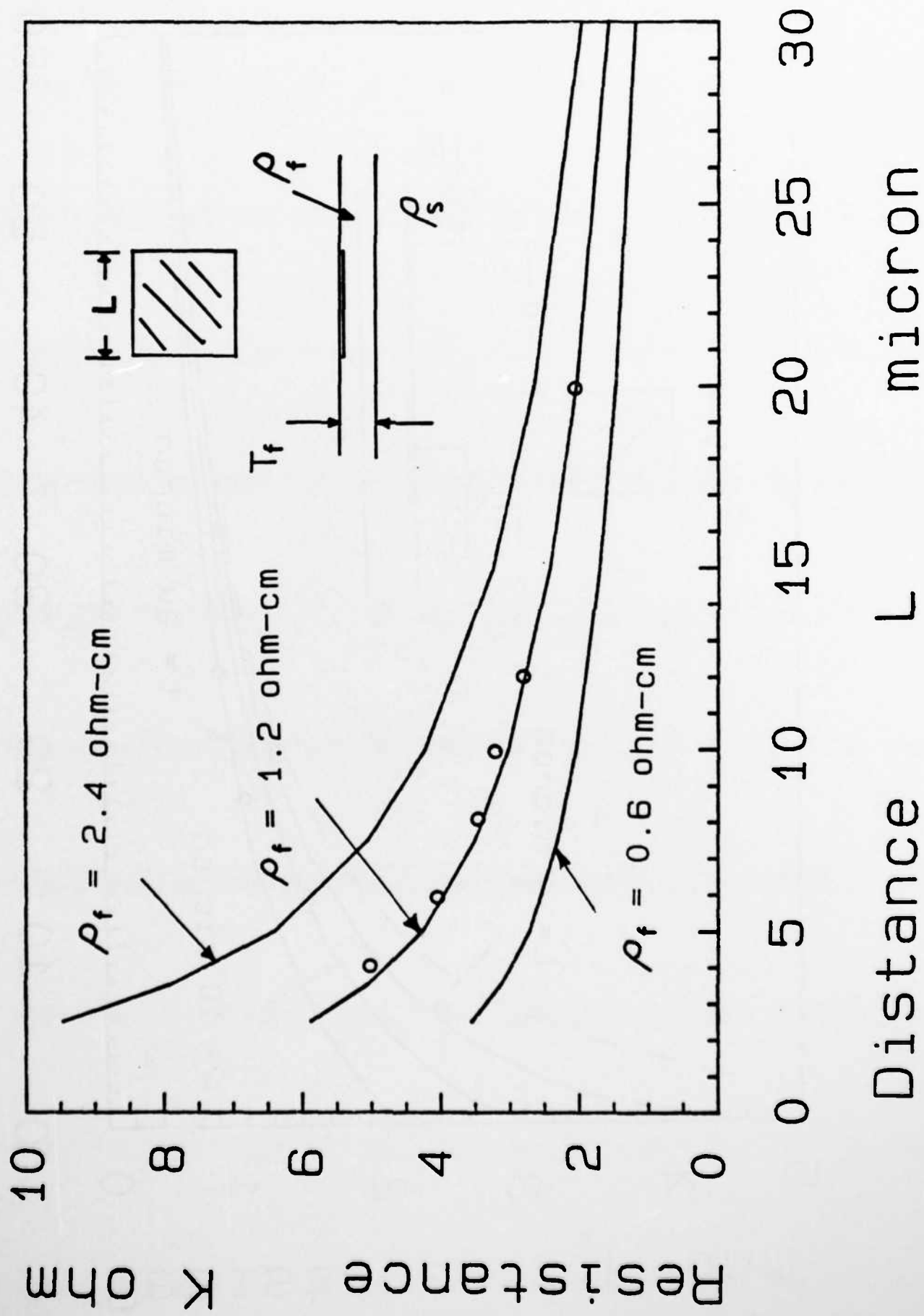
Figure Captions

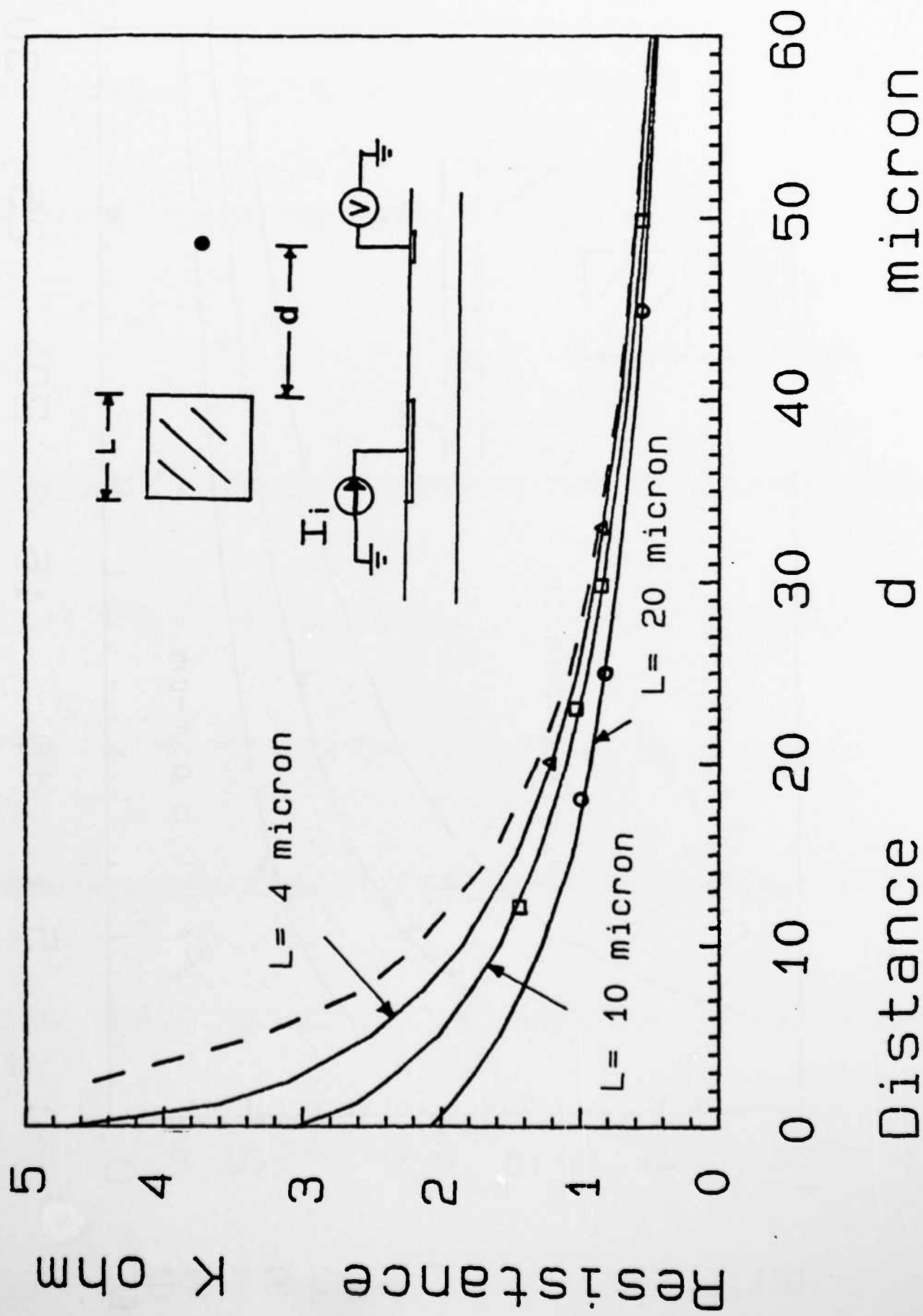
- (1) A input protection circuit used with an input inverter and its cross-section showing the p+/n and n+/p diodes.
- (2) The resistance between a square surface contact of side-length  $L$  and the backside as a function of both the resistivity of the field implant region " $\rho_f$ " and side-length. The substrate resistivity is 21 ohm-cm.
- (3) The transverse resistance of a surface contact as a function of the distance from the edge of the injector " $d$ " and the injector side-length " $L$ ". The broken line shows the transverse resistance for a homogeneous substrate with  $L = 10$  micron. The substrate and field implant region resistivities are 21 and 1.2 ohm-cm respectively.
- (4) The transverse resistance and the resistance to backside for a surface contact which completely encloses a region in its interior. The resistance is shown as a function of the thickness of the ring " $T$ " and the distance from its center " $d$ ". The substrate and field implant region resistivities are 21 and 1.2 ohm-cm respectively.
- (5) The effective substrate resistance due to a injector of side-length  $L=12$  micron which has a grounded guard-ring of side-length  $L=12$  micron located at a distance of 18 micron from the edge of the injector. The resistance is shown as a function of the distance  $d$  from the injector edge. The resistance due to the same injector with no guard-ring present is shown by the broken line. The substrate and field implant region resistivities are 21 and 1.2 ohm-cm respectively.

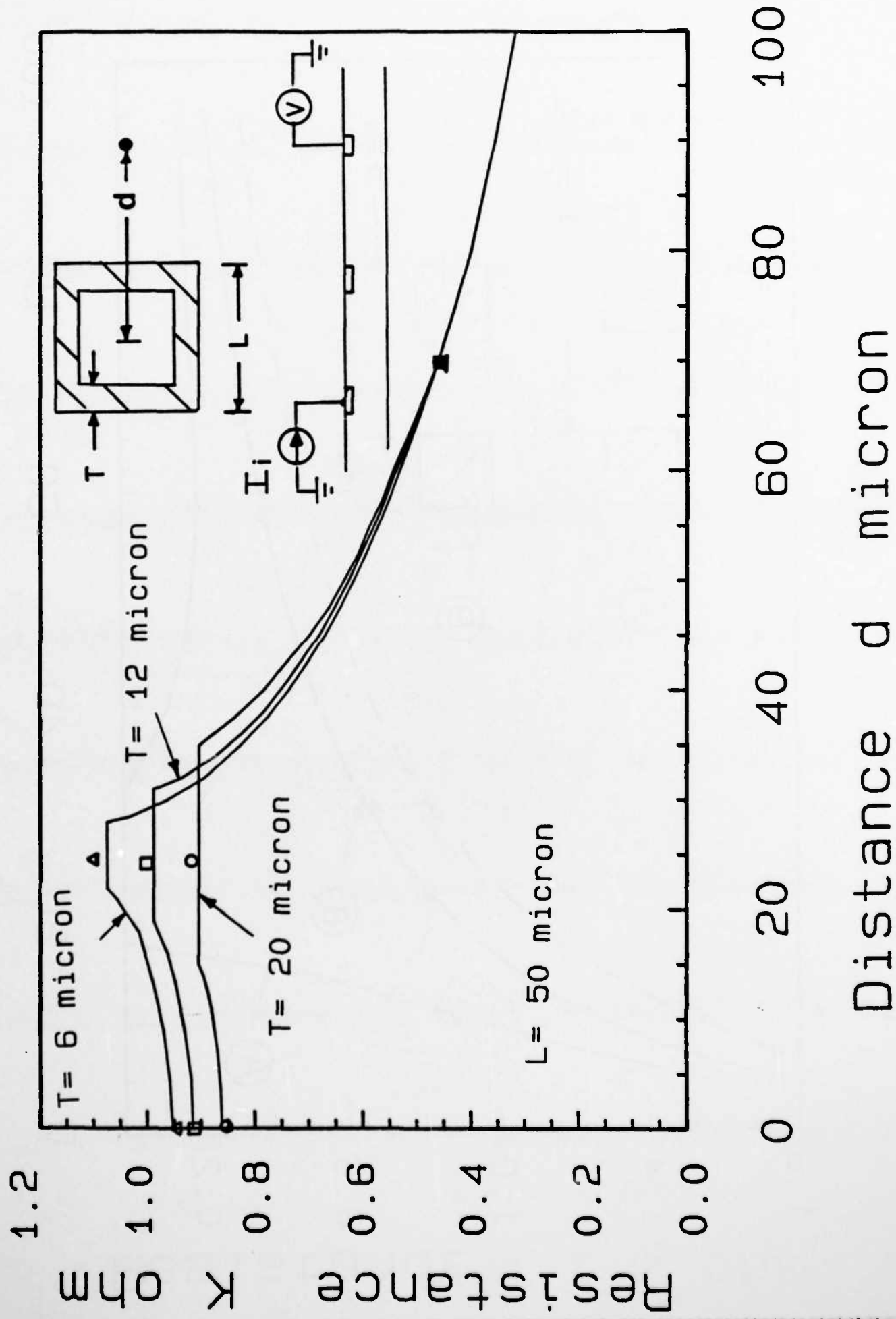
# GATE PROTECTION CIRCUIT

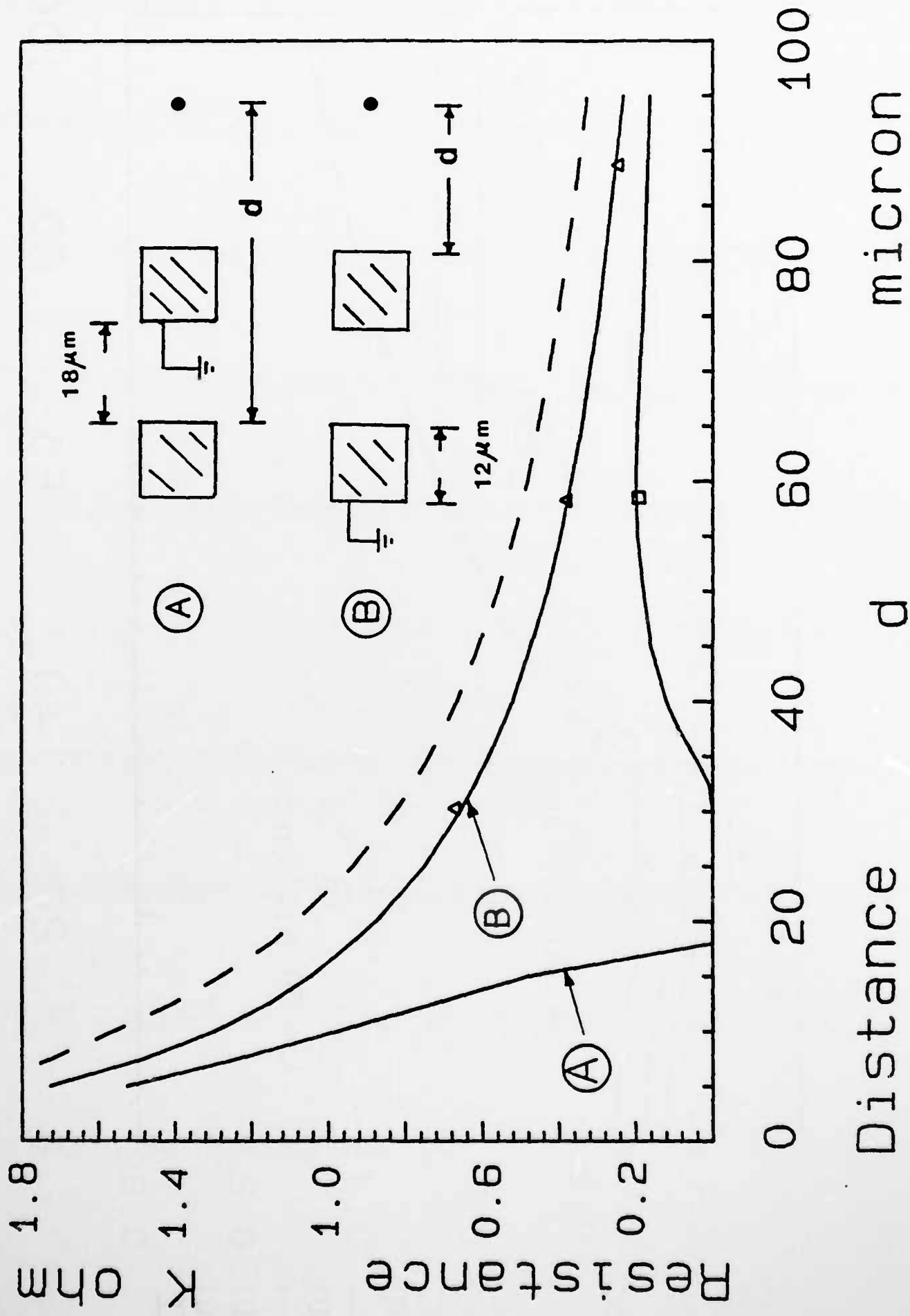












## COMPUTER ANALYSIS ON THE COLLECTION OF ALPHA-GENERATED CHARGE FOR REFLECTING AND ABSORBING SURFACE CONDITIONS AROUND THE COLLECTOR

K. TERRILL, C. HU and A. NEUREUTHER

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley,  
CA 94720, U.S.A.

(Received 3 June 1983)

**Abstract**—We present an analysis of the collection of alpha-particle generated charge by collectors surrounded by either uniform reflecting or uniform absorbing surfaces. These are the two extreme cases of any real condition that exists in IC's. The analysis of the upper limit of charge collection should be more useful for circuit design than the previously available lower limit. It is assumed that the charge transport is by diffusion. The effects of collector size,  $\alpha$ -particle energy, and the separation between the collector and the alpha track are studied. When the  $\alpha$ -particle strike is through the center of the collector, the difference in collected charge for the two cases is up to a factor of two. When the  $\alpha$ -particle strike does not pass through the collector, the difference is much greater. The collected charge scales approximately linearly with the collector side length.

### 1. INTRODUCTION

The discovery by May and Woods[1] that low levels of  $\alpha$ -particle radiation originating from packaging materials can cause soft errors in MOS dynamic RAM's has created considerable interest in the simulation of the transport and collection of carriers generated along an alpha-particle track. The soft error rate depends on the frequency of  $\alpha$ -particle strikes, the physics of charge generation, the collection process and the circuit vulnerability. This paper is mainly concerned with the collection process and its sensitivity to the boundary conditions which exist at the surface of the silicon chip.

Kirkpatrick has presented a thorough analysis of the collection phenomenon assuming a diffusion transport mechanism with a collector surrounded by a uniform absorbing surface[2]. This boundary condition can represent the surface under a depletion region which may be assumed to be absorbing (having infinite recombination velocity). Kirkpatrick was able to obtain an analytic solution since this boundary condition resulted in a uniform surface boundary condition. However, it is unknown how much more charge will be collected when the surrounding surface area is under the field oxide and may be assumed to be reflecting (having zero recombination velocity). For the purpose of circuit design, it is more important to know this upper bound of charge collection than the lower bound that Kirkpatrick derived.

The difficulty that occurs when one tries to account for both absorbing and reflecting surfaces is due to the mixed boundary conditions which now exist at the surface. These boundary conditions make it impossible to obtain an analytic solution to the diffusion equation. One attempt at accounting for the presence of both absorbing and reflecting surfaces relied on Monte Carlo simulations[3, 4].

This paper presents a solution to the mixed boundary problem of a collector surrounded by a uniform reflecting surface and compares it to the results of Kirkpatrick's boundary condition as the two extreme cases of any real condition in IC's. Another difference between Kirkpatrick's work and the present paper is that the former considered circular and line collectors while the present paper presents results for square collectors.

Recent work by Hsieh *et al.*[5, 6] has shown that for the case of an  $\alpha$ -particle passing through a depletion region the dominant transport mechanism may be drift rather than diffusion for a short period of time after the  $\alpha$ -particle strikes. In this case the present paper may still be able to help estimate the charge collected due to diffusion after this time period has elapsed[7].

### 2. MODEL

#### 2.1 Initial conditions and boundary conditions

The rate at which an energetic  $\alpha$ -particle loses energy as it travels in silicon has been calculated by Ziegler. His findings show that  $\alpha$ -particles of the same energy always take the same distance to stop and the rate of energy loss through ionization is a function of only the distance from the end of the track. The generated carriers thermalize in less than one picosecond to a distance of 0.1  $\mu\text{m}$  around the  $\alpha$ -particle path. The initial condition resulting from an  $\alpha$ -particle strike in silicon is thus a line of electron-hole pairs. The pair density increases and eventually peaks and drops to zero as the  $\alpha$ -particle slows down toward the end of the track.

There are two types of surfaces for the silicon. The surface under the field oxide can usually be considered a reflective boundary owing to the high quality of the  $\text{SiO}_2$ -Si interface and the heavy field doping

which produces a field opposing the flow of electrons toward the surface. The surface under the depletion region can be considered to be an absorbing boundary since the electric field there sweeps carriers away at a faster rate than they can diffuse to the surface. In the following discussions we will neglect the finite depth of the depletion region in our calculations. This reduces the diffusion problem to a planar geometry with mixed boundary conditions on the surface. We will present solutions for both a collector surrounded by a reflecting surface and a collector surrounded by an absorbing surface as the two extreme conditions which may exist in an IC.

For a rectangular collector surrounded by an absorbing surface the solution may be obtained easily from the method of images. The boundary condition for this case is that the concentration of carriers at the surface is zero over the entire surface [2].

For a rectangular collector surrounded by a reflecting surface we have solved the diffusion equation numerically. The boundary conditions corresponding to this case are:

(a) The concentration of carriers at the surface in the collector region is zero.

(b) The gradient of the carrier concentration at and normal to the surface is zero in the reflecting region.

## 2.2 Mathematical formulation

By restricting the transport mechanism to diffusion the flux through a collector surrounded by a reflecting surface after an  $\alpha$ -strike can be found by solving the diffusion eqn (1)

$$D\nabla^2\psi - \frac{\partial\psi}{\partial t} = -\frac{\psi}{\tau} \quad (1)$$

Where  $\psi$  is the concentration of carriers at any position  $\vec{r}$  and any time  $t$ ,  $D$  is the diffusion constant, and  $\tau$  is the recombination time in the substrate. By making the substitution  $\psi = N \exp(-t/\tau)$  the recombination term drops out of the diffusion equation yielding

$$D\nabla^2 N - \frac{\partial N}{\partial t} = 0. \quad (2)$$

In practice, the collection process ends in tens of nanoseconds—a time much shorter than the recombination time. As a result the difference between  $\psi$  and  $N$  will be ignored, i.e. recombination will be neglected, in this paper. It would be straightforward to include the effect of recombination, such as might be desirable for GaAs; then the collected charge will be dependent on  $\tau$ . The corresponding Green's function is found by solving the differential eqn (3)

$$D\nabla^2 G(\vec{r}, \vec{r}', t - t') - \frac{\partial G(\vec{r}, \vec{r}', t - t')}{\partial t} = \delta(\vec{r} - \vec{r}')\delta(t - t'). \quad (3)$$

Where  $G$  is a Green's function with an observation point  $\vec{r}$ , source location  $\vec{r}'$ , and boundary conditions that have not yet been specified. The solution of the diffusion equation is given quite generally by the Integral eqn (4)

$$N(\vec{r}, t) = D \int_0^t dt' \int da' \left\{ G(\vec{r}, \vec{r}', t - t') \frac{\partial N(\vec{r}', t')}{\partial n'} - N(\vec{r}', t') \times \frac{\partial G(\vec{r}, \vec{r}', t - t')}{\partial n'} \right\} + \int_V d\vec{r}' G(\vec{r}, \vec{r}', t) N(\vec{r}', 0). \quad (4)$$

Here  $N(\vec{r}', 0)$  is the initial excess carrier concentration generated by the  $\alpha$ -strike. Any initial excess carrier concentration can be used but for an  $\alpha$ -strike we will let the initial excess carrier concentration be a line of charge as described by Ziegler [8].

In order to solve the integral equation we must apply the boundary conditions specified and make an appropriate choice of a Green's function which allows us to solve the equation. It is convenient to choose a Green's function which has a zero gradient normal to the surface as this allows us to simplify the problem to finding  $N$  only on the collecting surface. This is accomplished quite easily through the method of images. It is clear that the problem of finding such a Green's function is equivalent to the problem of the original carrier concentration and an equal carrier concentration located at the mirror-image point above the plane defined by the position of the surface. The Green's function used is given in eqn (5).

$$G(\vec{r}, \vec{r}', t - t') = \{4\pi D(t - t')\} \times \exp \left\{ -\frac{(x - x')^2 + (y - y')^2}{4D(t - t')} \right\} \times \left\{ \exp \left\{ -\frac{(z - z')^2}{4D(t - t')} \right\} + \exp \left\{ -\frac{(z + z')^2}{4D(t - t')} \right\} \right\}. \quad (5)$$

The surface integral can be broken up into three distinct regions.

(a) The surface area located at the surface of the silicon in the collector region.

(b) The surface area located at the surface of the silicon in the reflecting region.

(c) The surface which encloses the hemisphere of infinite radius.

The surface integral over the hemispherical surface at infinity can be shown to be zero. Thus, the time integral will be zero for any finite time. If the bulk recombination is included then it is obvious that the time integral is zero even for an infinite time. Over the remaining surfaces the gradient of the Green's function is zero. This allows us to reduce the integral

equation by setting the dipole term equal to zero

Substituting this into eqn (8) yields:

$$N(\bar{r}, t) = D \int_0^t dt' \int_S d\bar{a}' G(\bar{r}, \bar{r}', t - t') \frac{\partial N(\bar{r}', t')}{\partial n'} + \int_V d\bar{r}' G(\bar{r}, \bar{r}', t) N(\bar{r}', 0). \quad (6)$$

$$\int_0^t dt' \int_S d\bar{a}' G(\bar{r}, \bar{r}', t - t') F(\bar{r}', t') = \int_V d\bar{r}' G(\bar{r}, \bar{r}', t) N(\bar{r}', 0). \quad (10)$$

By applying boundary condition (b) the integral over the reflecting surface is seen to be zero. Thus the only contributing portion of the integral is the integration over the collector region. This reduces the integral equation to:

$$N(\bar{r}, t) = D \int_0^t dt' \int_S d\bar{a}' G(\bar{r}, \bar{r}', t - t') \frac{\partial N(\bar{r}', t')}{\partial n'} + \int_V d\bar{r}' G(\bar{r}, \bar{r}', t) N(\bar{r}', 0). \quad (7)$$

where  $S$  represents the collector surface (region 1).

Finally to form an integral equation we need to restrict the observation point  $\bar{r}$  to lie on the surface in the collector region. Under the limit of approaching this boundary the concentration  $N(\bar{r}, t)$  must go to zero and the integral equation simplifies to:

$$D \int_0^t dt' G(\bar{r}, \bar{r}', t - t') \frac{\partial N(\bar{r}', t')}{\partial n'} = - \int_V d\bar{r}' G(\bar{r}, \bar{r}', t) N(\bar{r}', 0). \quad (8)$$

We are interested in obtaining the flux of carriers passing through the surface in the collector region. This flux is related to the carrier concentration by the relation:

$$F(\bar{r}', t') = -D \frac{\partial N(\bar{r}', t')}{\partial n'}. \quad (9)$$

Thus, the flux of carriers through the collecting surface can be found from the numerical solution of the integral equation over this area.

### 3. RESULTS

In the following subsections the total collected charge is presented for three different types of  $\alpha$ -particle strikes. The first case occurs when an  $\alpha$ -particle passes directly through the center of the collector. This case will be referred to as a "center hit". The "center hit" case has been published before but it will be briefly covered again here for completeness[9]. The second case occurs when an  $\alpha$ -particle passes through the edge of the collector. This case will be referred to as an "edge hit". The last case occurs when an  $\alpha$ -particle passes through the reflecting surface surrounding the collector. This case will be referred to as a "near miss". The results given are for a square collector with alpha strikes occurring normal to the surface. The figures to be presented give solutions for both a collector surrounded by a reflecting surface as an upper bound (solid line) and a collector surrounded by an absorbing surface as a lower bound (broken line). In both cases the diffusion constant used is

$$D = 25 \frac{\text{cm}^2}{\text{sec}}.$$

The essential difference between the two boundary conditions can be appreciated from the following two examples. Figure 1 shows the charge flux collected by

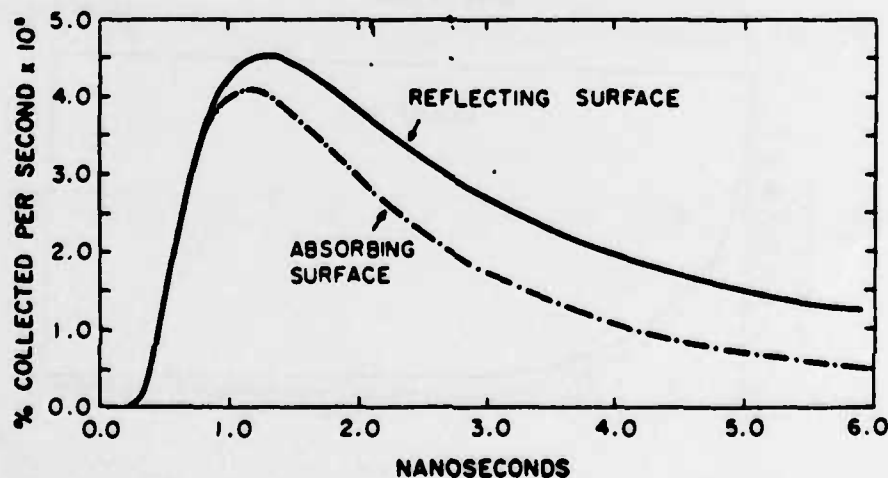


Fig. 1. The charge flux collected by a  $5 \times 5 \mu\text{m}$  collector from a point source located at the center of the collector and  $5 \mu\text{m}$  below the surface. In this and all other figures the reflecting surface is represented by a solid line while the absorbing surface is represented by a dashed line.



a  $5 \times 5 \mu\text{m}$  collector from a point source located  $5 \mu\text{m}$  below the surface. The boundary condition has negligible effect on the flux in the initial period. At times long enough for "reflected" carriers to be collected, the reflecting surface results in a higher flux than the absorbing surface. Figure 2 shows the collected charge as a function of location inside the collector area at several time instances. For the absorbing surface the plot shows that the charge always peaks in the center of the collector and is always a minimum at the edge. In the case of a reflecting surface the charge distribution has a local maximum at the edge of the collector which may be larger than the charge collected at the center. Clearly the larger amount of charge collected near the edge

of the collector is contributed by carriers reflected from the reflecting surface.

### 3.1 Center hit

We are interested in finding the collected charge which results from an  $\alpha$ -particle passing through the center of the collector. We split the problem into two parts in order to reduce the required computation time. First we find the impulse response of the system for a point source located at a distance  $Z$  below the collecting surface. The results of these calculations are shown in Fig. 3. It is important to note that for a center hit the total charge collected by the square collector from a point source is a function of only the ratio of the depth of the source  $Z$  and the side length

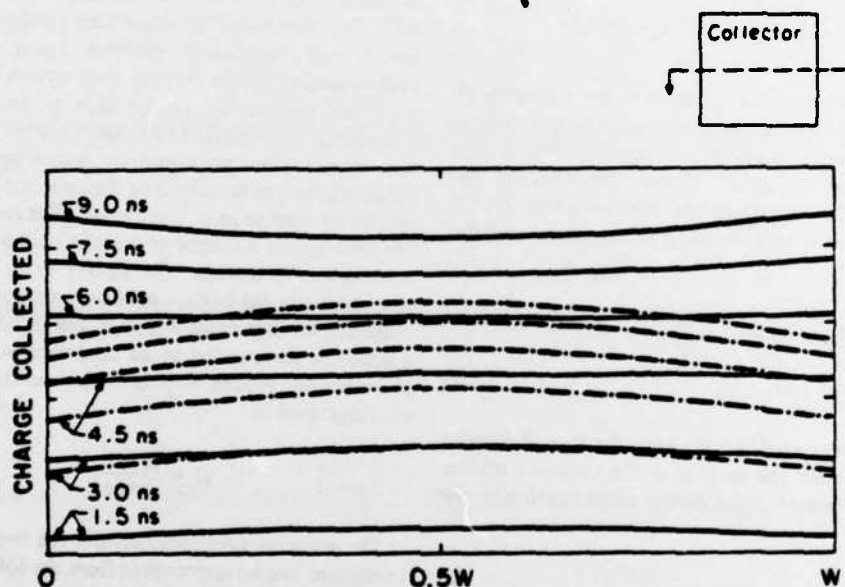


Fig. 2. The collected charge as a function of location inside the collector area at several points in time. The size of the collector is  $5 \times 5 \mu\text{m}$  and the point source is located at the center of the collector and  $7.5 \mu\text{m}$  below the surface.

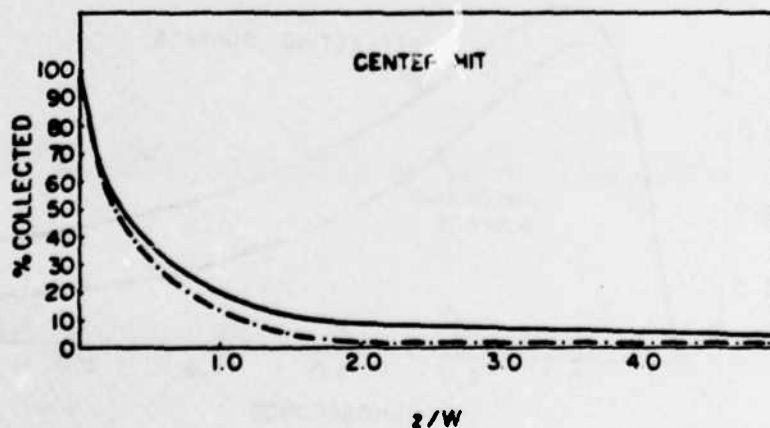


Fig. 3. The impulse response for a center hit with a collector of side length  $W$  and a point source at distance  $Z$  below the collecting surface.

of the collector  $W$ . This allows us to calculate the charge collected by collectors of many different sizes form the same impulse response. Finally to calculate the collected charge for a given carrier distribution along the  $\alpha$ -particle track we only need to convolve the given distribution [8] with the impulse response shown in Fig. 3.

The total charge collected as a function of alpha energy for a square collector with side lengths of 2.5, 5.0, 7.5, and 10.0  $\mu\text{m}$  is shown in Fig. 4. As would be expected the effect of the reflecting surface is to increase the charge collected. For  $\alpha$ -particles striking in the center of the collector it is at most an increase of 40% for 5 MeV  $\alpha$ -particle energies and lower. For higher  $\alpha$ -particle energies or for  $W < 2.5 \mu\text{m}$  the increase can be as much as 80%.

The total charge collected as a function of collector size is shown in Fig. 5. As noted by Kirkpatrick [2] the

collected charge scales linearly with the radius for the absorbing surface. For the reflecting surface the results is similar except that for low  $\alpha$ -particle energies and when the length of the side is larger than approximately three microns the collected charge decrease slightly less rapidly than a linear curve would predict. In contrast, the charge stored on the storage capacitor of a dynamic RAM may scale with the second power of feature length. Figure 5 also shows that the reflecting surface results in about 60% more charge collected than the absorbing surface for  $W < 3 \mu\text{m}$ .

### 3.2 Edge hit

The impulse response for an initial point source of carriers lying a distance  $Z$  below the edge of the collector is given in Fig. 6. Once again it is important to note that the collected charge is only a function of

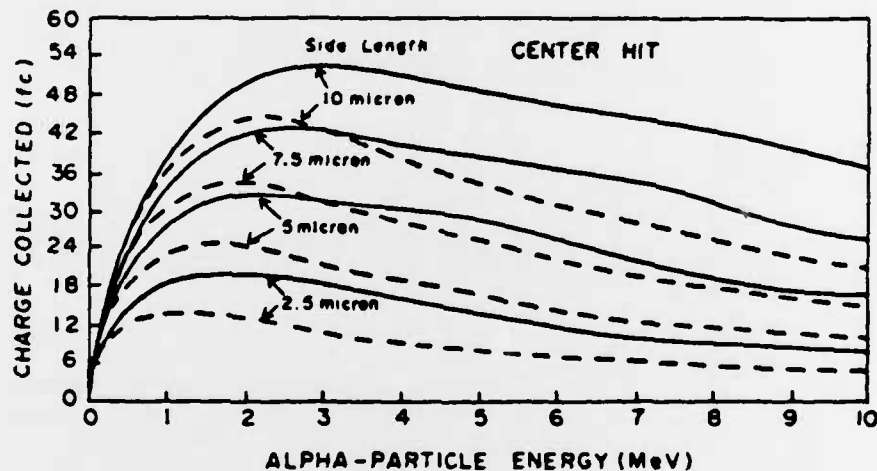


Fig. 4. Total charge collected for a center hit as a function of  $\alpha$ -particle energy for  $\alpha$ -particles striking the center of collectors with side lengths of 2.5, 5.0, 7.5 and 10.0  $\mu\text{m}$ .

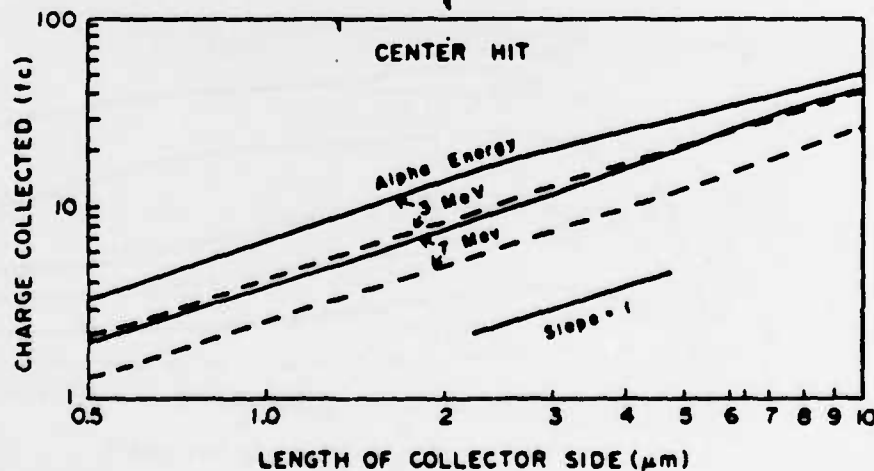


Fig. 5. Total charge collected as a function of collector side length for  $\alpha$ -particles of 3 and 7 MeV striking the center of the collector.

the ratio of the distance from the surface to the source and the side length of the collector.

The total collected charge as a function of alpha energy for a square collector with side lengths of 2.5, 5.0, 7.5 and 10.0  $\mu\text{m}$  is shown in Fig. 7. As would be expected the reflecting surface has a greater effect on the collected charge in the case of an edge hit than in the case of a center hit. The collected charge for a square collector of side length  $2W$  surrounded by an absorbing surface is by coincidence about equal to the collected charge for a square collector of side length  $W$  surrounded by a reflecting surface.

### 3.3 Near miss

Figure 8 shows the locations of these strikes in comparison to the direct hit and edge hit. It is important to note in this case that the collected

charge is not only a function of the ratio of the depth of the source and the side length of the collector  $Z/W$ . We must also specify the distance  $X$  from the edge of the collector to the point where the source projects onto the surface. This dependence also scales so that the collected charge may be expressed in terms of the ratios  $X/W$  and  $Z/W$ . Figure 9 shows the impulse responses for near misses for  $X/W = 0.5, 1$  and  $1.5$ . We note that for a collector surrounded by an absorbing surface the collection efficiency peaks at some distance away from the surface and that this distance increases as we move away from the collector. This is due to the fact that carriers generated very close to the surface have a high probability of being absorbed by the absorbing surface before reaching the collector. For the case of a collector surrounded by a reflecting surface the maximum

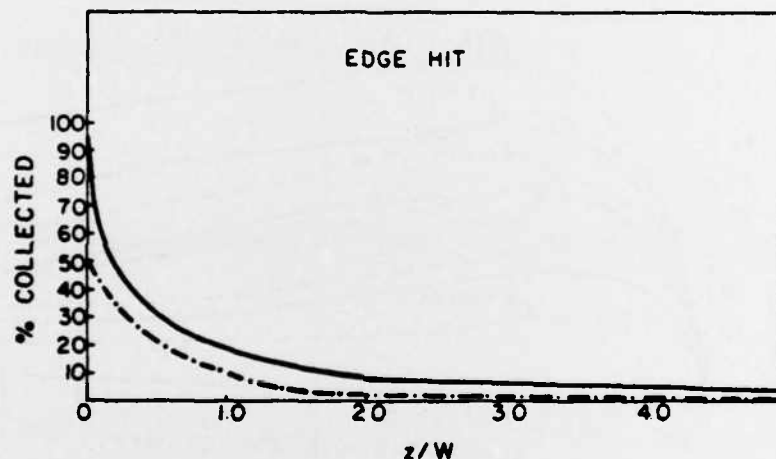


Fig. 6. The impulse response for an edge hit with a collector side length  $W$  and a point source at distance  $Z$  below the collecting surface.

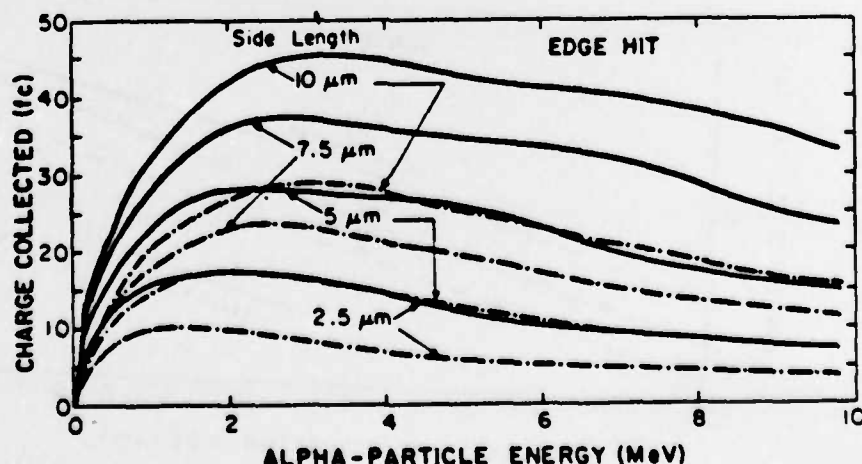


Fig. 7. Total charge collected for an edge hit as a function of  $\alpha$ -particle energy for  $\alpha$ -particles striking the edge of collectors with side lengths of 2.5, 5.0, 7.5 and 10.0  $\mu\text{m}$ . The reflecting surface is represented by a solid line while the absorbing surface is represented by a dashed line.

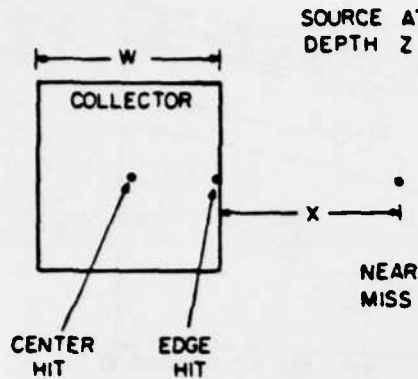


Fig. 8. Location of center hit, edge hit and near miss strikes in relation to the collector.

collection efficiency occurs at the surface. Thus, the two solutions diverge from each other as they approach the surface.

Figure 10 compares the charge collected for a center hit, an edge hit and a near miss as a function of  $\alpha$ -particle energy using a square collector of side length  $5\mu\text{m}$ . We can see in this figure that as the strike location moves away from the collector the collection efficiency decays rapidly for the case of a collector surrounded by an absorbing surface. In comparison, for a collector surrounded by a reflecting surface the collection efficiency decays much more slowly. It is also important to note that as the  $\alpha$ -particle strike moves further from the center of the collector the peak in the collection efficiency occurs at higher  $\alpha$ -particle energies.

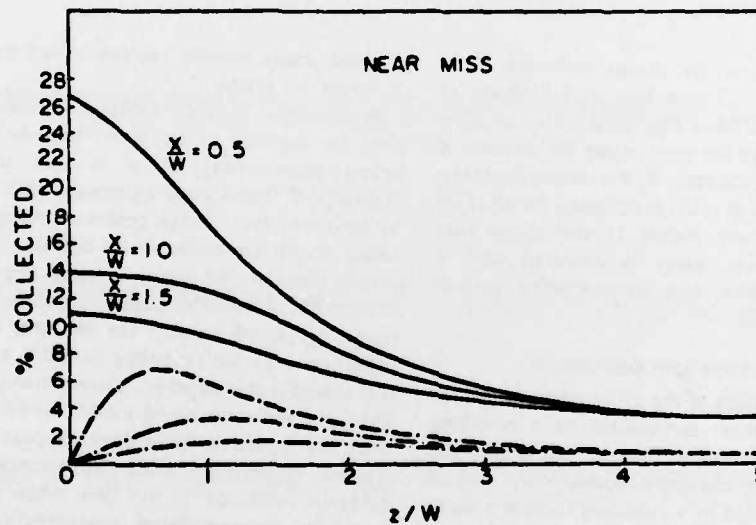


Fig. 9. The impulse response for near misses at  $X/W = 0.5, 1.0$  and  $1.5$  with a collector side length  $W$  and a point source at distance  $Z$  below the collecting surface.

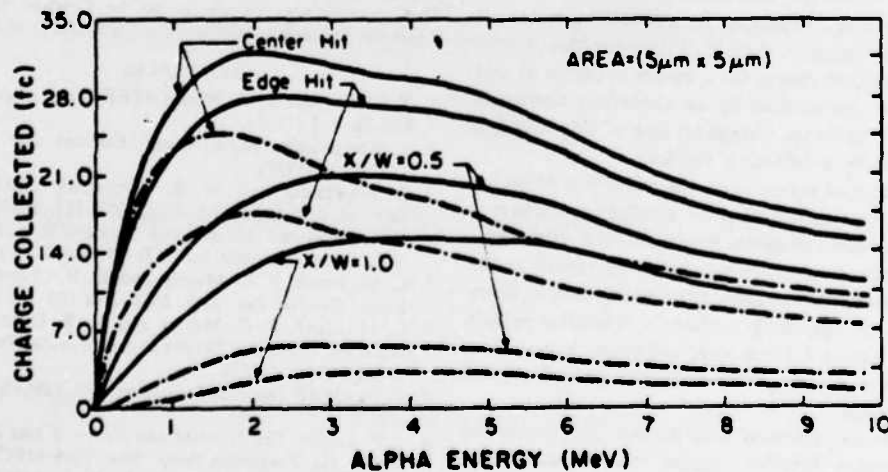


Fig. 10. Total charge collected for a center hit, an edge hit and a near miss as a function of  $\alpha$ -particle energy using a square collector of side length  $5\mu\text{m}$ . The reflecting surface is represented by a solid line while the absorbing surface is represented by a dashed line.

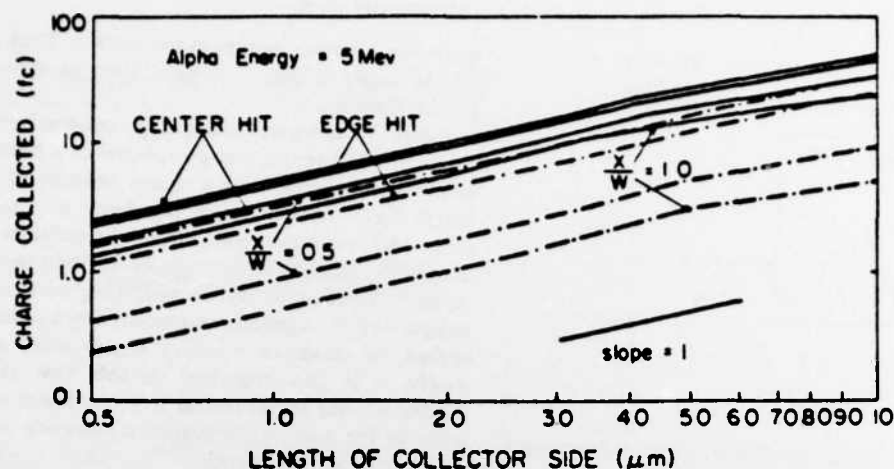


Fig. 11. Total charge collected for a center hit, an edge hit and a near miss as a function of collector side length for an  $\alpha$  energy of 5 MeV.

Figure 11 compares the charge collected for a center hit, edge hit and near miss as a function of collector side length on a Log-Log scale for an  $\alpha$  energy of 5 MeV. For the near misses the distance  $X$  is also scaled so that the ratio  $X/W$  remains constant. At small side lengths the charge collected for all cases is approximately linear. Figure 11 also shows that about 7 times more charge is collected with a reflecting surface than with an absorbing surface when  $X/W = 1.0$ .

#### 4. CONCLUSION AND DISCUSSION

A numerical solution of the diffusion equation for the case of a collector surrounded by a reflecting surface has been presented. For a "center hit" the results show that the charge collected for the case of a collector surrounded by a reflecting surface is up to two times that for the case of one surrounded by an absorbing surface.

For an "edge hit" the difference in collected charge between a collector surrounded by an absorbing surface and one surrounded by a reflecting surface is more pronounced. A rule of thumb for this case is that the collected charge for a square collector of side lengths  $2W$  surrounded by an absorbing surface is equal to the collected charge for one of side length  $L$  surrounded by a reflecting surface.

In the case of a near miss the collection efficiency for a collector surrounded by a reflecting surface is shown to decrease much slower than it does for a collector surrounded by an absorbing region as the strike location moves away from the collector. Compared to an absorbing surface, a reflecting surface results in about 7 times more collected charge when an  $\alpha$  strikes at one collector length away from the collector edge.

The results presented here do not depend on the recombination lifetime because recombination has been neglected. This is a good approximation for the collection of  $\alpha$ -generated charge in silicon. The anal-

ysis can easily include recombination as might be desirable for GaAs.

While diffusion based analysis is probably incomplete for the case of center hit because of the funneling phenomena [5, 6] it is still useful. Sai-Halasz [3, 4] found good agreement with experiment by assuming that charge generated outside the funneling length are collected by diffusion. Hu [7] suggested that all the carriers along the track drift toward the junction by a funneling length during the funneling period so that the problem of diffusion collection after the funneling period is equivalent to that solved in this paper for a lower-energy  $\alpha$ -particle. Thus, the diffusion based model, which is complete when the  $\alpha$ -particle strike does not pass through the junction depletion region, can also be useful for diffusion collection in this case when properly adjusted for the new initial conditions after the funneling phenomena.

**Acknowledgements**—The authors wish to thank W. G. Oldham and F. J. Henley for many informative discussions. This research was sponsored by the Defense Advanced Research Projects Agency contract N00039-K-0251.

#### REFERENCES

1. T. C. May and M. H. Woods, *IEEE Trans. Electron Dev.* ED-26, 2-9 (1979).
2. S. Kirkpatrick, *IEEE Trans. Electron Dev.* ED-26, 1742-1753 (1979).
3. G. Sai-Halasz and M. R. Wordeman, *IEEE Trans. Electron Dev. Lett.* EDL-1(10), 211-213 (1982).
4. G. A. Sai-Halasz, M. R. Wordeman and R. H. Dennard, *IEEE Trans. Electron Dev.* ED-29, 725-731 (1982).
5. C. M. Hsieh, P. C. Murley and R. R. O'Brien, *IEEE Trans. Electron Dev. Lett.* EDL-2(4) 103-105 (1981).
6. C. M. Hsieh, P. C. Murley and R. R. O'Brien, *Proc. IEEE Int. Reliability Physics Symp. Orlando, Florida*, pp. 38-42 7 April (1981).
7. C. Hu, *IEEE Trans. Electron Dev. Lett.* EDL-3(2), 31-34 (1982).
8. J. F. Ziegler, *The Stopping and Range of Ions in Matter*, Vol. 4: He, Pergamon Press, New York (1977).
9. K. W. Terrill, C. Hu and A. R. Neureuther, *Solid St. Electron.* 26, 15-18 (1983).